US009466124B2

(12) United States Patent
Akenine-Moller

(10) Patent No.: US 9,466,124 B2
(45) Date of Patent: Oct. 11, 2016

(54) **COMPRESSION USING INDEX BITS IN MSAA**

(71) Applicant: **Intel Corporation**, Santa Clara, CA (US)

(72) Inventor: **Tomas G. Akenine-Moller**, Lund (SE)

(73) Assignee: **Intel Corporation**, Santa Clara, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 45 days.

(21) Appl. No.: **14/536,756**

(22) Filed: **Nov. 10, 2014**

(65) **Prior Publication Data**

US 2016/0133029 A1 May 12, 2016

(51) **Int. Cl.**

G06T 1/20	(2006.01)
G09G 5/02	(2006.01)
G09G 5/06	(2006.01)
G09G 5/39	(2006.01)
G09G 5/14	(2006.01)
G06F 11/00	(2006.01)
G06F 9/00	(2006.01)
G06T 9/00	(2006.01)

(52) **U.S. Cl.**

CPC . ***G06T 9/00*** (2013.01); ***G06T 1/20*** (2013.01); *G06T 2200/12* (2013.01); *G06T 2210/08* (2013.01)

(58) **Field of Classification Search**

CPC combination set(s) only.
See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,559,954 A *	9/1996	Sakoda	G09G 5/02 345/501
8,687,010 B1 *	4/2014	Battle	G06T 1/60 345/552
8,743,142 B1 *	6/2014	Hutchins	G09G 5/02 345/552
2008/0117221 A1 *	5/2008	Hutchins	G06F 9/3867 345/506
2013/0176211 A1 *	7/2013	Inada	G06T 1/60 345/156
2014/0333662 A1 *	11/2014	Akenine-Moller	G09G 3/2003 345/604
2014/0347385 A1 *	11/2014	Akenine-Moller	G06T 5/002 345/611
2015/0235384 A1 *	8/2015	Nystad	G06T 11/001 345/582

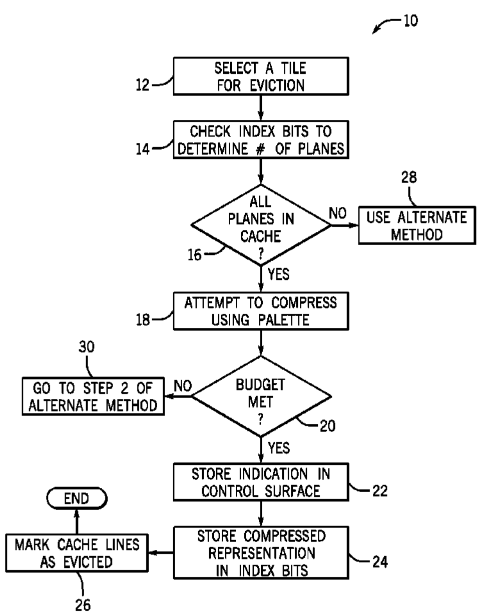
* cited by examiner

Primary Examiner — Wesner Sajous
(74) *Attorney, Agent, or Firm* — Trop, Pruner & Hu, P.C.

(57) **ABSTRACT**

A palette compressed representation may be stored in the index bits, when that is possible. The savings are considerable in some embodiments. In uncompressed mode, the data uses 2304 (2048+256) bits, and in compressed mode, the data uses 1280 bits. However, with this technique, the data only uses the index bits, (e.g. 256 bits) with a 5:1 compression improvement over the already compressed representation, and with respect to the uncompressed representation it is a 9:1 compression ratio.

30 Claims, 12 Drawing Sheets



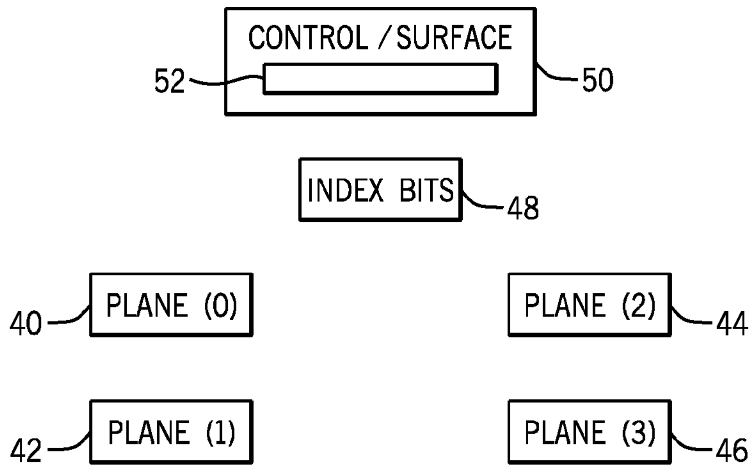

CONTROL / SURFACE
52
50
INDEX BITS
48
PLANE (0)
40
PLANE (2)
44
PLANE (1)
42
PLANE (3)
46

FIG. 1

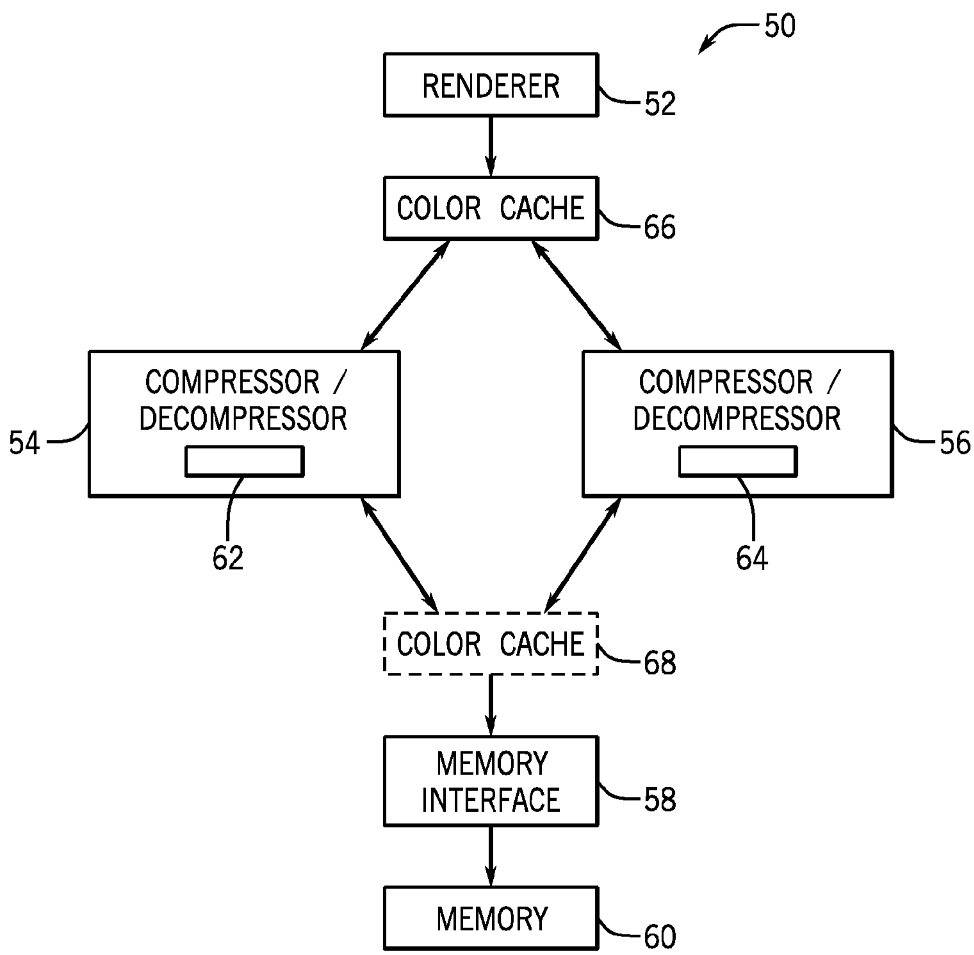

50
RENDERER
52
COLOR CACHE
66
COMPRESSOR / DECOMPRESSOR
54
62
COMPRESSOR / DECOMPRESSOR
56
64
COLOR CACHE
68
MEMORY INTERFACE
58
MEMORY
60

FIG. 3

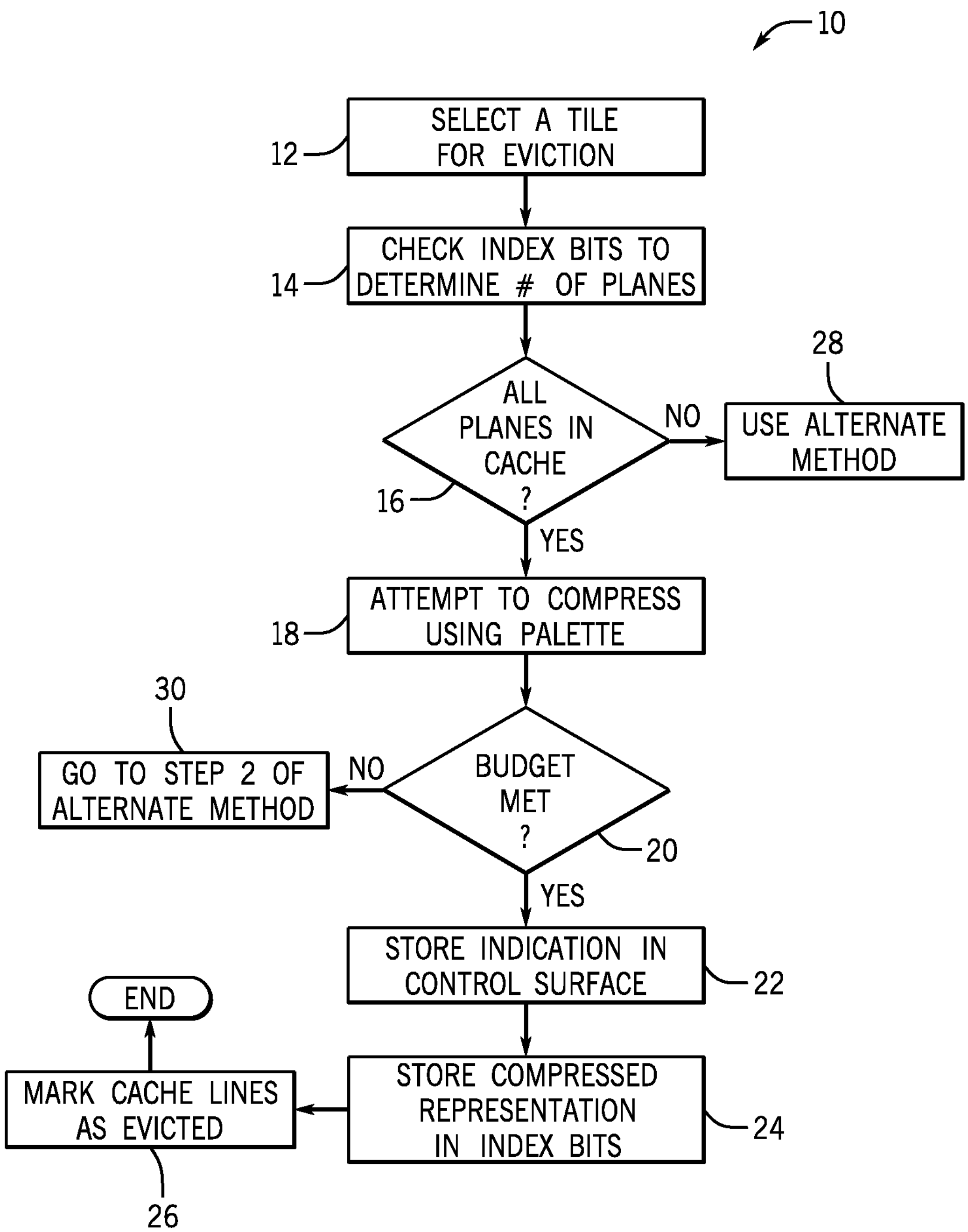

10
SELECT A TILE FOR EVICTION
12
CHECK INDEX BITS TO DETERMINE # OF PLANES
14
ALL PLANES IN CACHE ?
16
NO
USE ALTERNATE METHOD
28
YES
ATTEMPT TO COMPRESS USING PALETTE
18
BUDGET MET ?
20
NO
GO TO STEP 2 OF ALTERNATE METHOD
30
YES
STORE INDICATION IN CONTROL SURFACE
22
STORE COMPRESSED REPRESENTATION IN INDEX BITS
24
MARK CACHE LINES AS EVICTED
26
END

FIG. 2

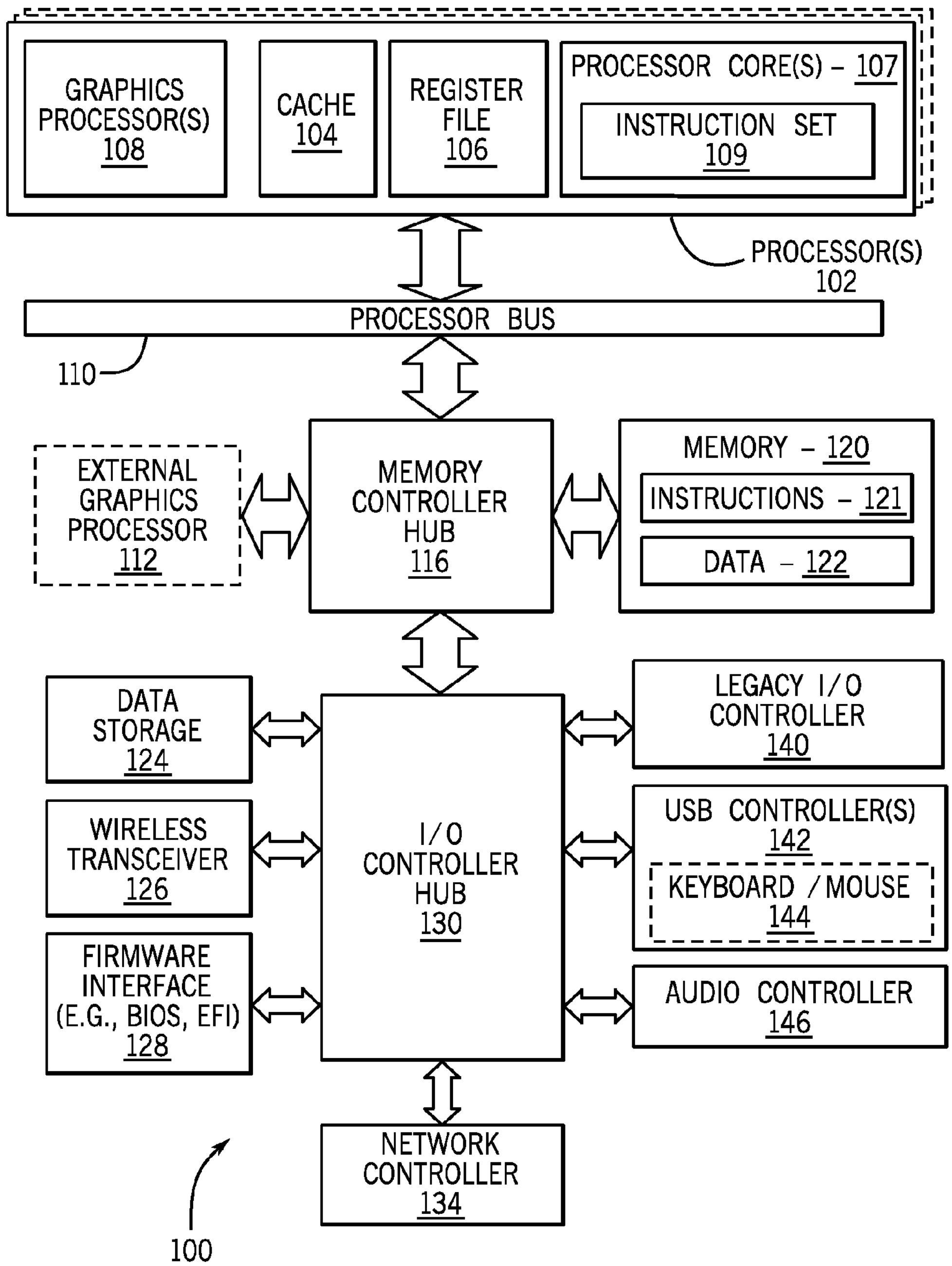

GRAPHICS PROCESSOR(S) 108
CACHE 104
REGISTER FILE 106
PROCESSOR CORE(S) – 107
INSTRUCTION SET 109
PROCESSOR(S) 102
PROCESSOR BUS
110
EXTERNAL GRAPHICS PROCESSOR 112
MEMORY CONTROLLER HUB 116
MEMORY – 120
INSTRUCTIONS – 121
DATA – 122
DATA STORAGE 124
WIRELESS TRANSCEIVER 126
FIRMWARE INTERFACE (E.G., BIOS, EFI) 128
I/O CONTROLLER HUB 130
LEGACY I/O CONTROLLER 140
USB CONTROLLER(S) 142
KEYBOARD / MOUSE 144
AUDIO CONTROLLER 146
NETWORK CONTROLLER 134
100

FIG. 4

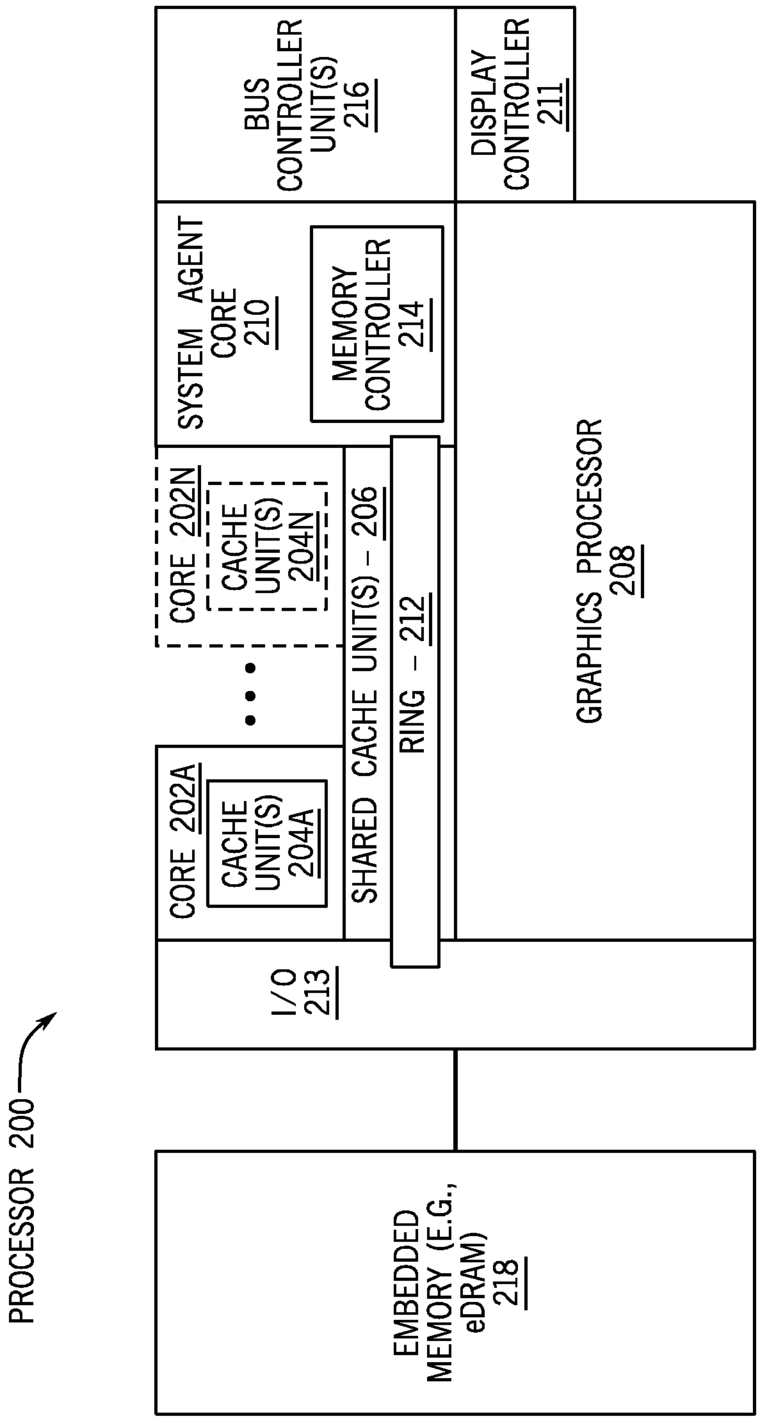

PROCESSOR 200
BUS CONTROLLER UNIT(S) 216
DISPLAY CONTROLLER 211
SYSTEM AGENT CORE 210
MEMORY CONTROLLER 214
CORE 202N
CACHE UNIT(S) 204N
CORE 202A
CACHE UNIT(S) 204A
SHARED CACHE UNIT(S) – 206
RING – 212
GRAPHICS PROCESSOR 208
I/O 213
EMBEDDED MEMORY (E.G., eDRAM) 218

FIG. 5

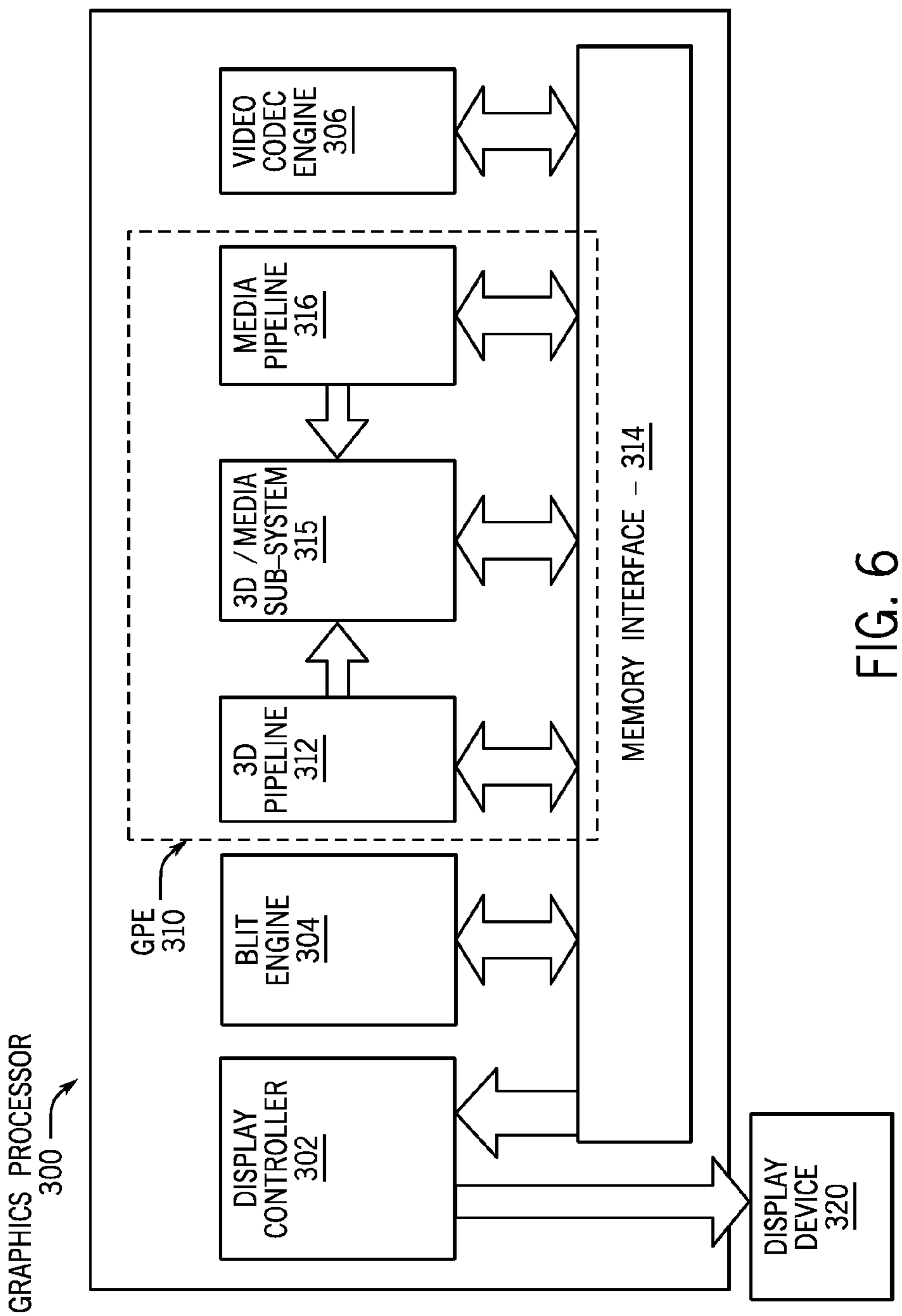

GRAPHICS PROCESSOR 300
DISPLAY CONTROLLER 302
BLIT ENGINE 304
GPE 310
3D PIPELINE 312
3D /MEDIA SUB-SYSTEM 315
MEDIA PIPELINE 316
VIDEO CODEC ENGINE 306
MEMORY INTERFACE – 314
DISPLAY DEVICE 320

FIG. 6

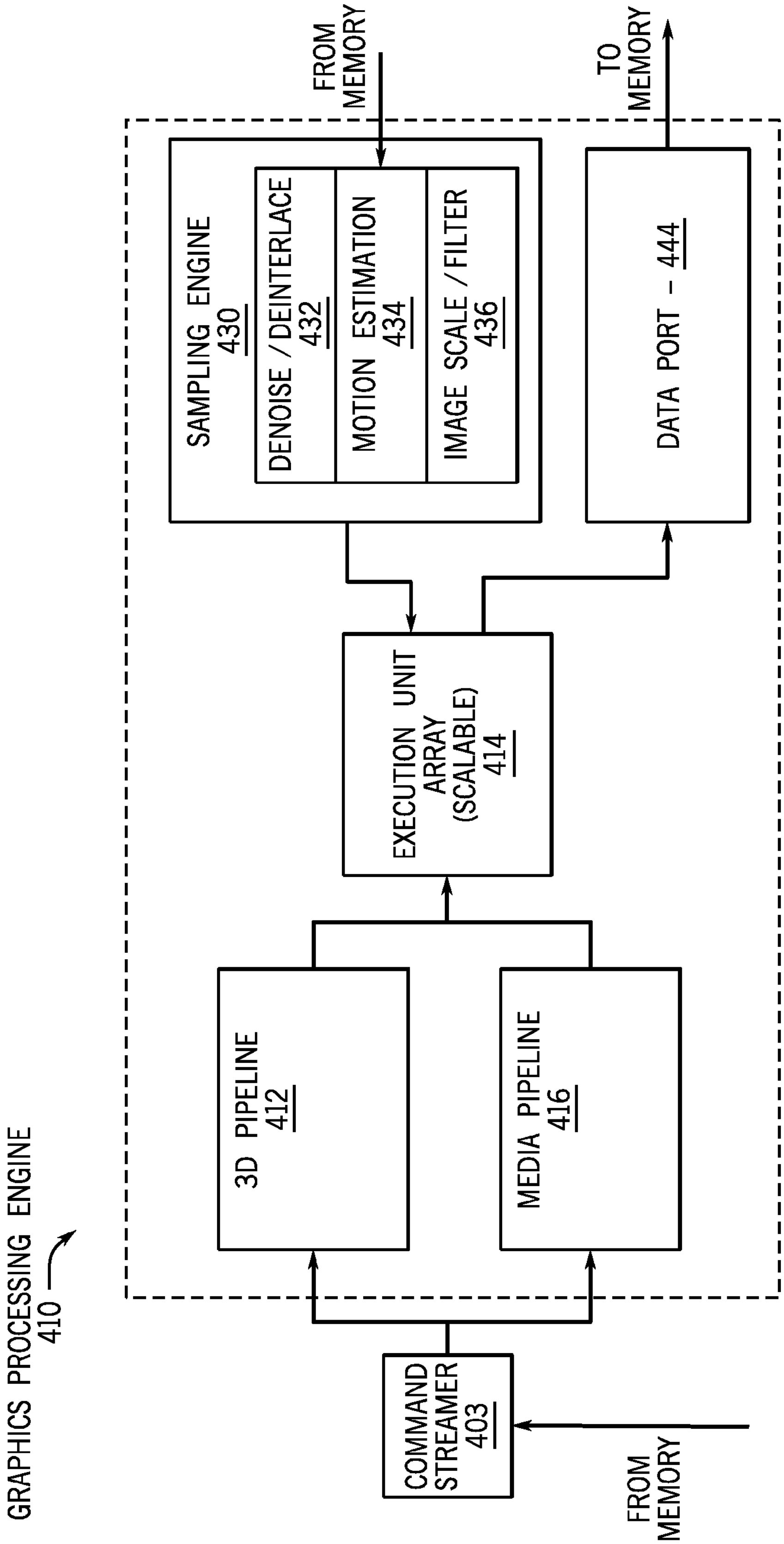
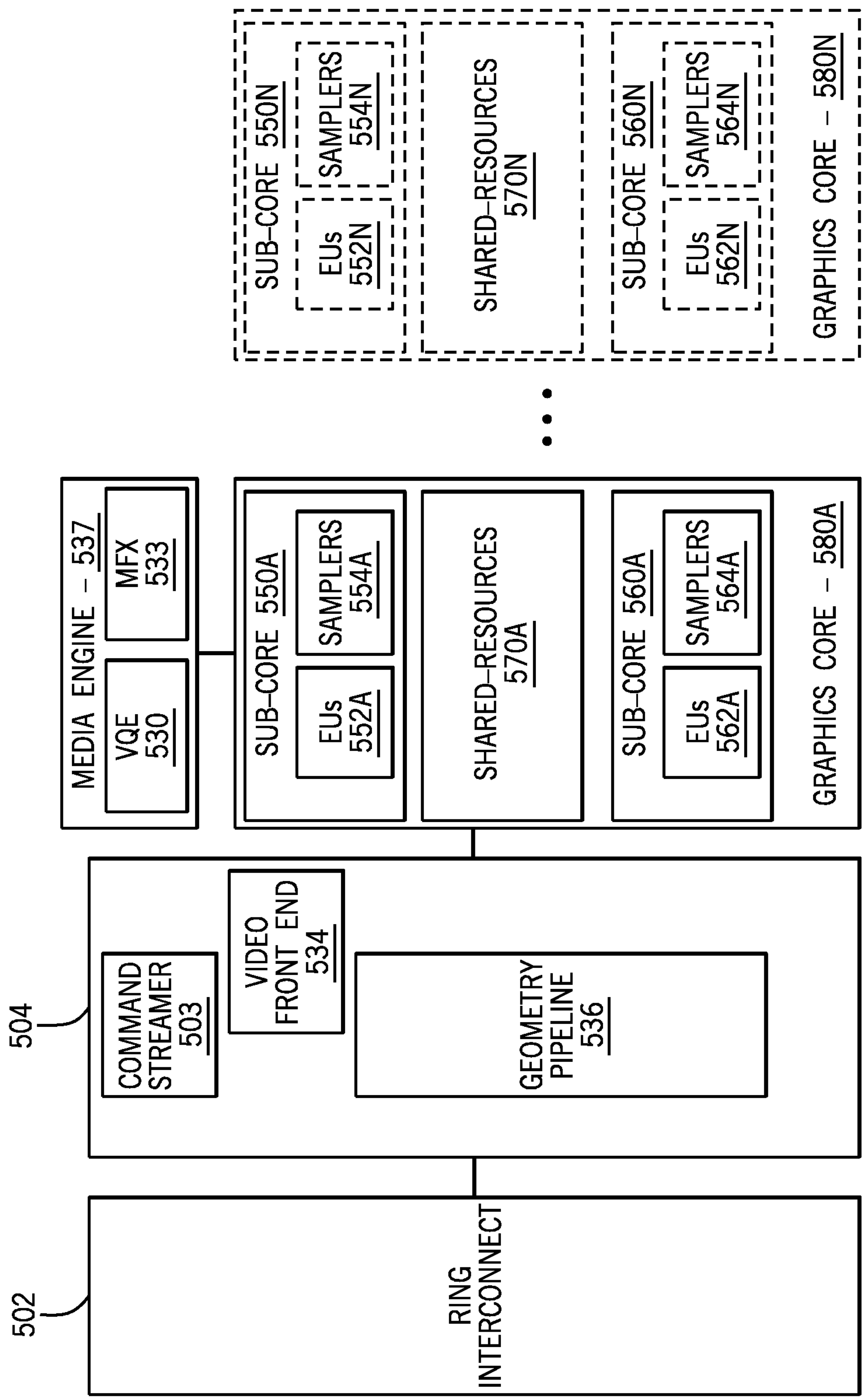

FROM MEMORY
TO MEMORY
SAMPLING ENGINE 430
DENOISE / DEINTERLACE 432
MOTION ESTIMATION 434
IMAGE SCALE / FILTER 436
DATA PORT - 444
EXECUTION UNIT ARRAY (SCALABLE) 414
3D PIPELINE 412
MEDIA PIPELINE 416
GRAPHICS PROCESSING ENGINE 410
COMMAND STREAMER 403
FROM MEMORY

FIG. 7



502
RING INTERCONNECT
504
COMMAND STREAMER 503
VIDEO FRONT END 534
GEOMETRY PIPELINE 536
MEDIA ENGINE – 537
VQE 530
MFX 533
SUB-CORE 550A
EUs 552A
SAMPLERS 554A
SHARED-RESOURCES 570A
SUB-CORE 560A
EUs 562A
SAMPLERS 564A
GRAPHICS CORE – 580A
SUB-CORE 550N
EUs 552N
SAMPLERS 554N
SHARED-RESOURCES 570N
SUB-CORE 560N
EUs 562N
SAMPLERS 564N
GRAPHICS CORE – 580N

FIG. 8

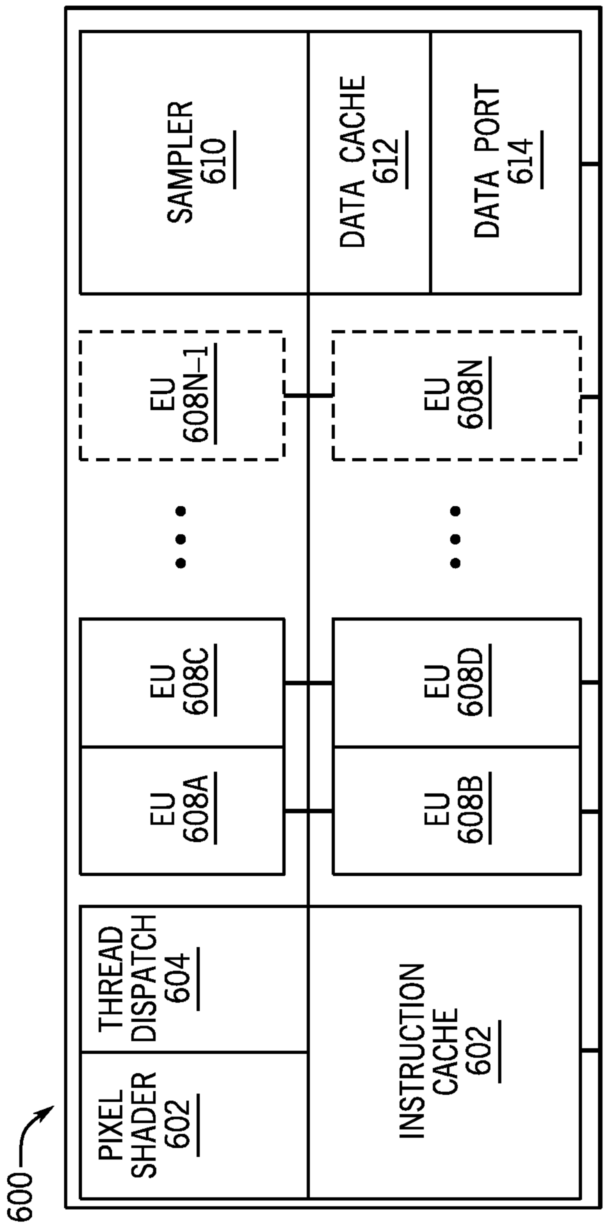

600
PIXEL SHADER 602
THREAD DISPATCH 604
INSTRUCTION CACHE 602
EU 608A
EU 608B
EU 608C
EU 608D
EU 608N-1
EU 608N
SAMPLER 610
DATA CACHE 612
DATA PORT 614

FIG. 9

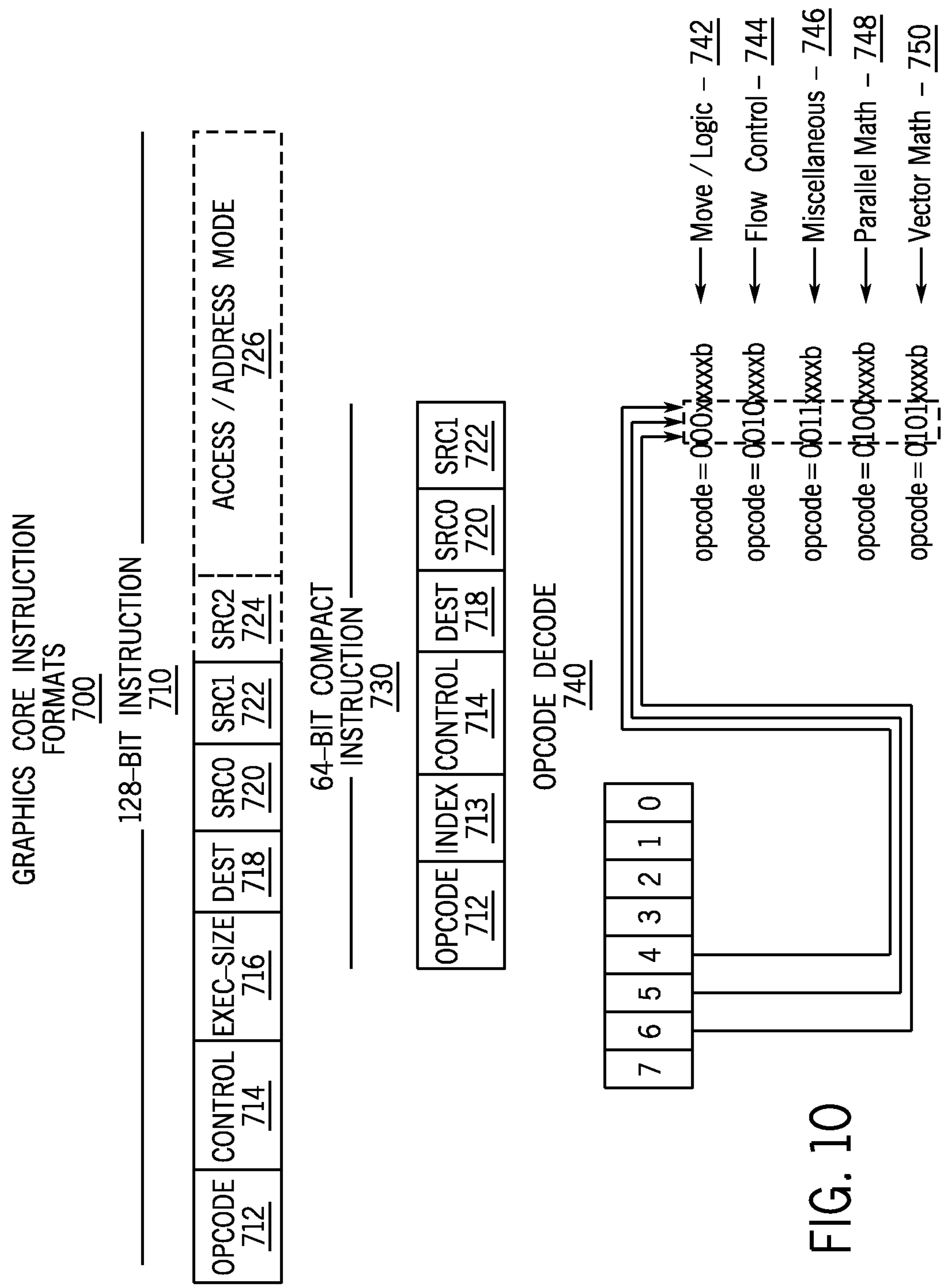

GRAPHICS CORE INSTRUCTION FORMATS 700
128–BIT INSTRUCTION 710
OPCODE 712
CONTROL 714
EXEC–SIZE 716
DEST 718
SRC0 720
SRC1 722
SRC2 724
ACCESS / ADDRESS MODE 726
64–BIT COMPACT INSTRUCTION 730
OPCODE 712
INDEX 713
CONTROL 714
DEST 718
SRC0 720
SRC1 722
OPCODE DECODE 740
7 6 5 4 3 2 1 0
opcode = 0000xxxxb
opcode = 0010xxxxb
opcode = 0011xxxxb
opcode = 0100xxxxb
opcode = 0101xxxxb
Move / Logic – 742
Flow Control – 744
Miscellaneous – 746
Parallel Math – 748
Vector Math – 750

FIG. 10

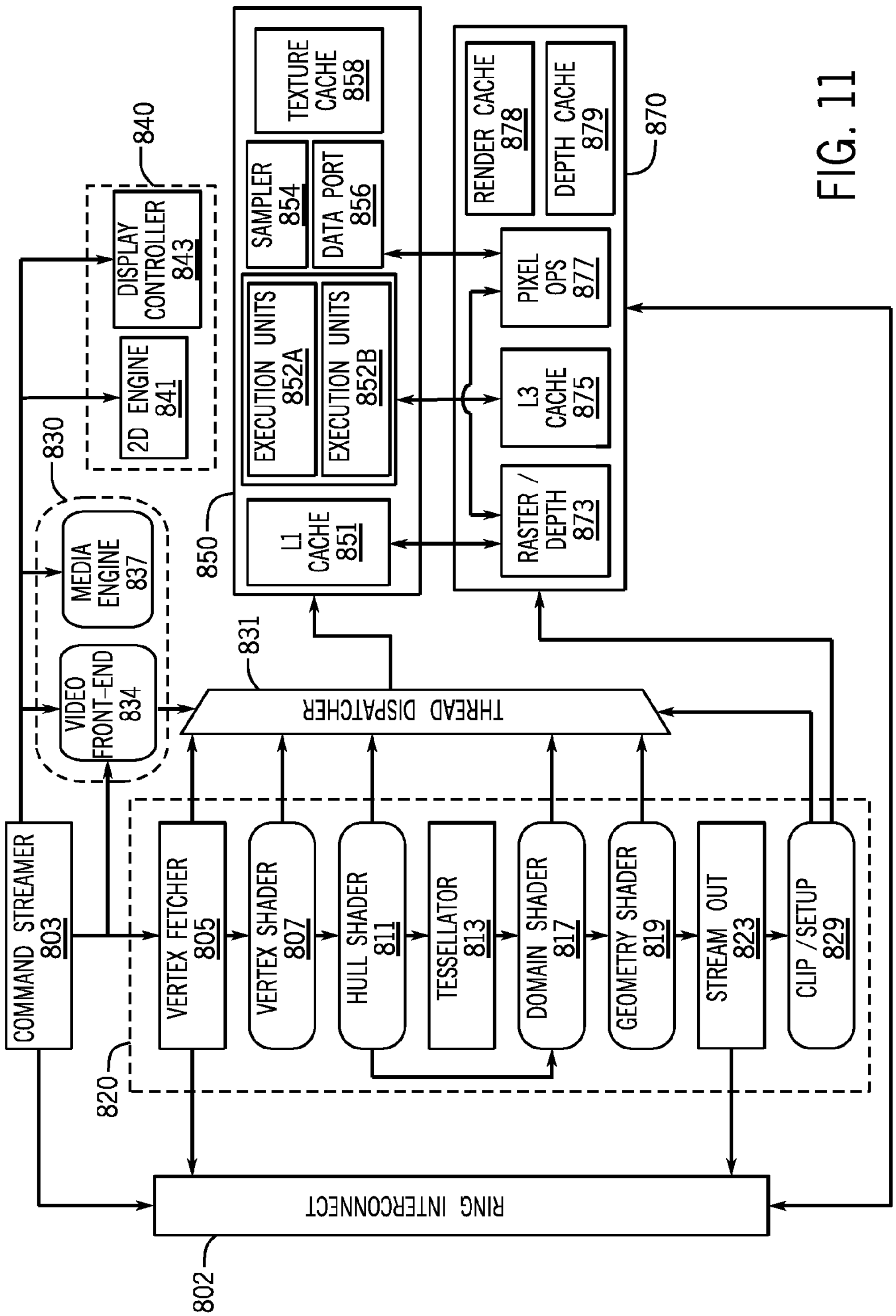

COMMAND STREAMER 803
RING INTERCONNECT
802
820
VERTEX FETCHER 805
VERTEX SHADER 807
HULL SHADER 811
TESSELLATOR 813
DOMAIN SHADER 817
GEOMETRY SHADER 819
STREAM OUT 823
CLIP /SETUP 829
830
VIDEO FRONT-END 834
MEDIA ENGINE 837
831
THREAD DISPATCHER
840
2D ENGINE 841
DISPLAY CONTROLLER 843
850
L1 CACHE 851
EXECUTION UNITS 852A
EXECUTION UNITS 852B
SAMPLER 854
DATA PORT 856
TEXTURE CACHE 858
870
RASTER / DEPTH 873
L3 CACHE 875
PIXEL OPS 877
RENDER CACHE 878
DEPTH CACHE 879

FIG. 11

FIG. 12A

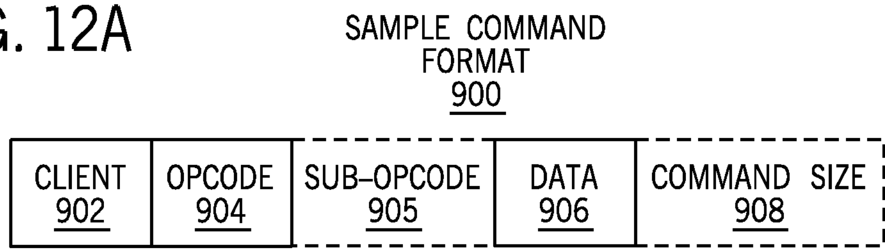
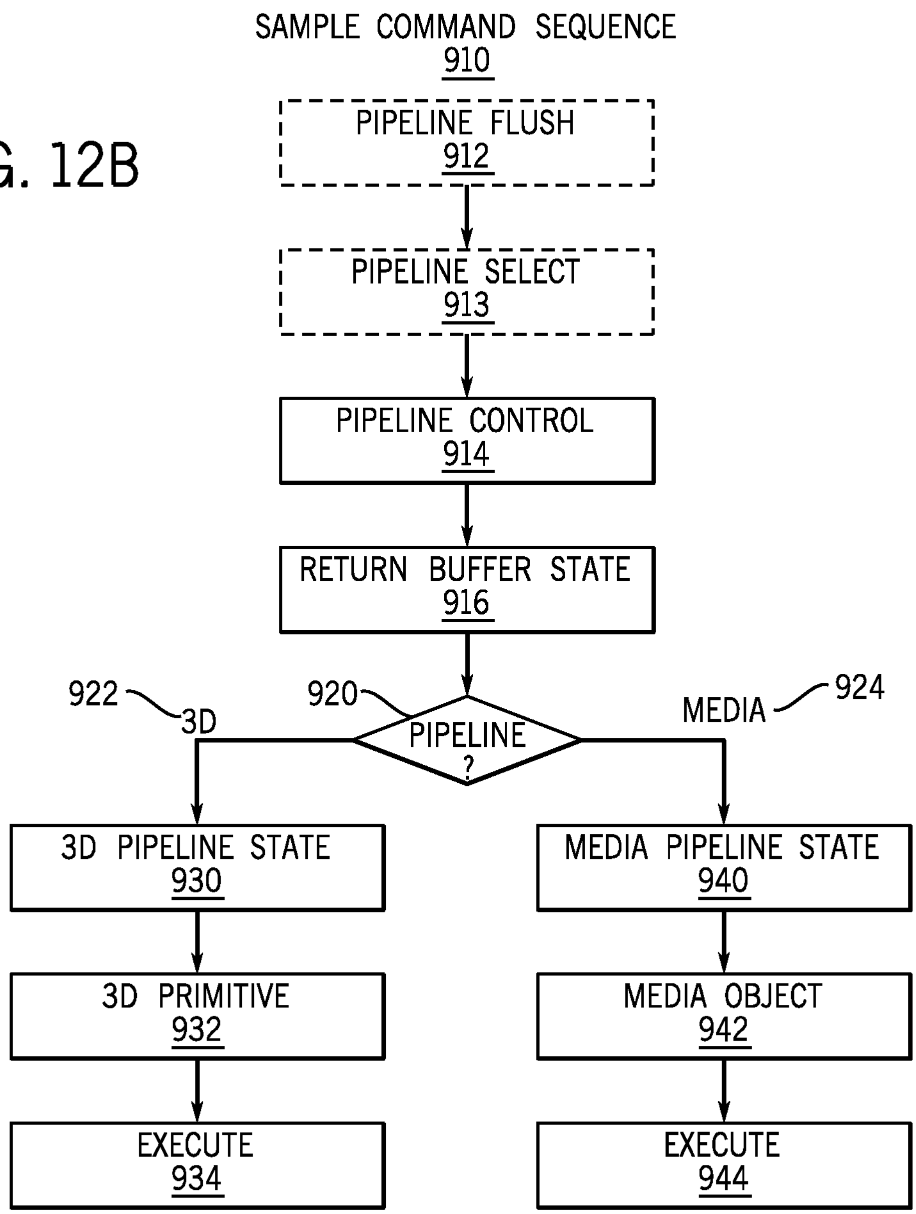


FIG. 12B



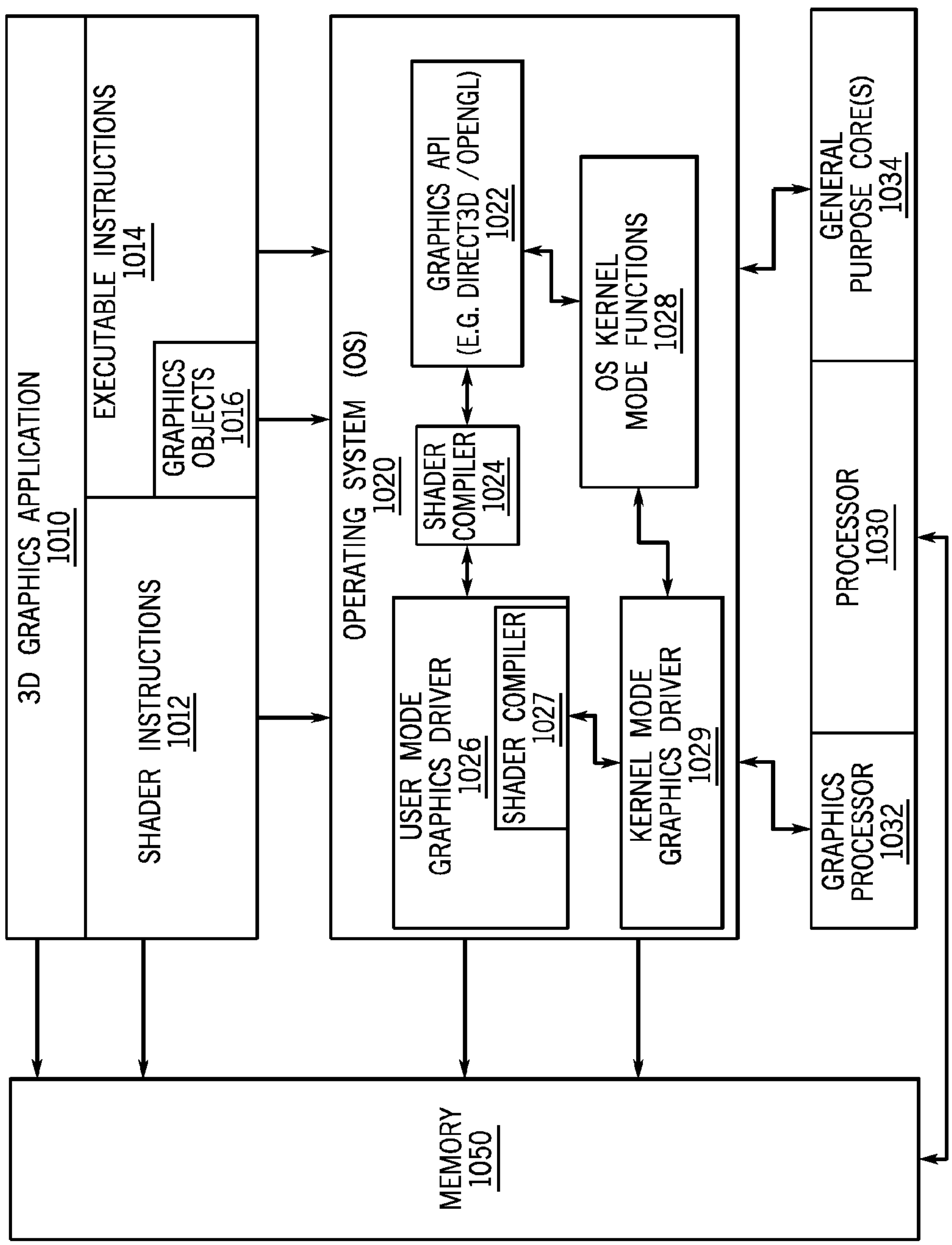

3D GRAPHICS APPLICATION
1010
EXECUTABLE INSTRUCTIONS
1014
GRAPHICS OBJECTS
1016
SHADER INSTRUCTIONS
1012
OPERATING SYSTEM (OS)
1020
GRAPHICS API
(E.G. DIRECT3D /OPENGL)
1022
SHADER COMPILER
1024
OS KERNEL MODE FUNCTIONS
1028
USER MODE GRAPHICS DRIVER
1026
SHADER COMPILER
1027
KERNEL MODE GRAPHICS DRIVER
1029
GENERAL PURPOSE CORE(S)
1034
PROCESSOR
1030
GRAPHICS PROCESSOR
1032
MEMORY
1050

FIG. 13

COMPRESSION USING INDEX BITS IN MSAA

BACKGROUND

Multi-sampling anti-aliasing (MSAA) improves the visual quality of the images rendered using a graphics processor. In MSAA, multiple visibility samples, color, and depth samples are used per pixel. However, during the rendering of a triangle, the pixel shader is only executed once per pixel. A common use case is 4× MSAA, where there are four times as many samples per pixel compared to the usual case of one sample per pixel. The color buffer bandwidth may increase by a factor of N for N× MSAA in a naïve setting. Therefore, the MSAA color buffers are compressed.

One way to compress MSAA color buffers is to split the colors into N different planes for N× MSAA, and always fill plane **0** first and continue with the other planes after that. Each pixel also needs log 2(N) bits, called index bits, to "point" to a color in the N different planes. For example, with 4× MSAA, each sample needs 2 index bits to point to a color being located in one of the 4 planes. For a tile of 8×4 pixels with 4 samples per pixel (and two bits per sample), 256 index bits (2*4*8*4) are needed.

If a pixel is completely inside the triangle being rendered, then all samples will get the same color, and the index bits will then be all zeroes (because they will all point to color plane **0**) for all samples, and nothing will be stored in the remaining planes. In many cases, only plane **0** will be used, and the remaining planes will be used more and more depending on the complexity of the geometry being rendered into a tile (e.g., 8×4 pixels).

If a color plane is empty for a certain tile, then there is no need to write that content to memory on a cache evict, and there is no need to read it either, when the color content for a tile is requested. Hence, this is a type of compression. However, one can also apply compression to the color planes when they are evicted from the color cache.

For each tile, a small number of bits (e.g., 4) are stored to indicate which state the tile is in. These bits can be used to indicate that the tile is cleared, or that plane **0** is compressed, while plane **1** is not, etc. For one render target, all these bits are called the control surface.

BRIEF DESCRIPTION OF THE DRAWINGS

Some embodiments are described with respect to the following figures:

FIG. **1** is a depiction of a color buffer for one embodiment;

FIG. **2** is a flow chart for one embodiment;

FIG. **3** is a schematic depiction of a graphics rendering system according to one embodiment;

FIG. **4** is a block diagram of a data processing system according to one embodiment;

FIG. **5** is a block diagram of the processor shown in FIG. **4** according to one embodiment;

FIG. **6** is a block diagram of the graphics processor of FIG. **4** according to one embodiment.

FIG. **7** is a block diagram of a graphics processing engine according to one embodiment;

FIG. **8** is a block diagram of a graphics processor according to another embodiment;

FIG. **9** illustrates thread execution logic for one embodiment;

FIG. **10** is a block diagram of a graphics processor execution unit instruction format according to one embodiment;

FIG. **11** is a block diagram of another embodiment of a graphics processor;

FIG. **12**A is a block diagram of a graphics processor command format according to one embodiment;

FIG. **12**B is a block diagram of a graphics processor command sequence according to one embodiment; and

FIG. **13** is a graphics software architecture for one embodiment.

DETAILED DESCRIPTION

Assume an 8×4 tile of pixels with 4 samples per pixel, and that first a blue triangle is rendered covering the entire tile. Then a green triangle is rendered, and it cuts through the tile, so that one third of the tile is green and the rest is blue. With the MSAA approach above, two color planes would be needed, because some pixels will contain at least one green sample and at least one blue sample. For an 8×4 tile with 4× MSAA the number of index bits sum to 256 bits (2 bits per sample times 4 samples per pixel times 8×4 pixels), and one plane stores 1024 bits for RGBA8 (32 bits per RGBA pixel times 8×4). Two planes take 2048 bits of storage. With 50% compression, the compressed version would use 1024 bits for both planes, i.e., the total cost is 1280 (1024+256) bits.

Assume a certain compression algorithm, such as a delta compression method, where the smallest color value is stored per tile, and the difference against the smallest color is encoded using as few bits as possible per channel. Such a method may be combined with a palette-based method, so that for each tile, compressing with both these methods is attempted followed by picking the one that compresses to the smallest usable size. This makes it possible to efficiently handle tiles with a few colors (e.g., 8) whose values are widely different. However, simple cases, such as the one above, will still use 1280 bits. A palette-based compression method stores, say, M colors (called a palette), and then each sample stores ceil ($\log_2(M)$) bits that are used to point to one of the colors in the palette. A palette-based compression method is better at handling widely different colors (than a delta compression method) because the colors in the palette can take on any value. A delta-based compression method usually stores a base color, and then stores differences in relation to the base color. However, this means that the differences must be small, which in turns indicates that all the colors in the tile must be rather similar. This is the reason why delta-based and palette-based compression methods are good to combine.

Referring to FIG. **1**, a depiction of a color buffer according to one embodiment includes a control surface **50**, index bits **48** and in this case 4 planes, plane **0**, plane **1**, plane **2** and plane **3** identified as **40**, **42**, **44** and **46** respectively.

The control surface's bits may be used in a special mode to indicate that none of the planes are used, and that a compressed color representation resides inside the index bits of the tile. A compressed color representation for the tile is stored in the tile's index bits, and this is marked in the control surface bits for the tile.

Then only the 256 index bits (in addition to the control surface bits) need to be accessed in the case of 4× MSAA. Assume that 4 bits per tile are used for the control surface, and if these bits are 1111 (all ones), then the tile is in this special mode in one embodiment. In FIG. **1**, the bits **52** may be used for the special mode.

With 4× MSAA, and R8G8B8A8 format, two 32-bit colors are stored, which sums to 64 bits, and there are 128 index bits, that is, 1 bit per sample for 8×4 pixels each with 4 samples per pixel. This sums to 192 bits. This 2-color palette may be stored in the index bits (256 bits per tile, and so 192 fits), so that cases such as the green/blue tile described above may be handled, and a significant amount of memory traffic is saved.

In general, there are M colors in the palette, and each sample needs to be able to point to any of the M colors. Each sample needs M combinations. If M is a power of 2, then each sample needs $\log_2(M)$ bits for its combinations.

A palette compressed representation may be stored in the index bits, when that is possible. The savings are considerable in some embodiments. In uncompressed mode, the data uses 2304 (2048+256) bits, and in compressed mode, the data uses 1280 bits. However, with this technique, the data only uses the index bits, (e.g. 256 bits) with a 5:1 compression improvement over the already compressed representation, and with respect to the uncompressed representation it is a 9:1 compression ratio.

Another variant (called method B) first uses three bits per pixel to indicate the following situations:

Bits	Situation Indicated by Bits
000	all samples in the pixel point to color 0
001	all samples in the pixel point to color 1
010	all samples in the pixel point to color 2
011	the samples in the pixel point to either color 0 or color 1
100	the samples in the pixel point to either color 0 or color 2
101	the samples in the pixel point to either color 1 or color 2
110	the samples in the pixel point to either color 0, color 1, or color 3
111	unused - or indicate that all samples in the pixel are cleared.

The “simple pixels”, pointing to only one color, correspond to combinations 000, 001, and 010. The “medium pixels”, pointing to two colors, correspond to bit combinations 011, 100, and 101, while bit combination 110 is called a “complex pixel”, because it points to three colors.

Then 96 (32*3) bits are stored for the three colors for the palette, and then three bits per pixel (3*32=96 bits) are stored as well. This leaves 64 (256−96*2) remaining bits.

The simple pixels do not require any more storage, but the medium and complex ones do. The medium pixels require one bit per sample, which in this example, amounts to 4 bits per medium pixel. If no complex pixels are required, then 16 (64/4) medium pixels can be stored. Given that there are 32 pixels in the tile, this is quite a lot.

A complex pixel needs to be able to point to three different colors. The 64 unused bits may be spent on the medium and complex pixels, and if these bits are sufficient for the current tile, the compression succeeds, and otherwise, it fails, and then some other compression method is used, or the data may be sent uncompressed. So, with 4× MSAA, a pixel has 4 samples, and each sample needs three combinations. Hence, one can store 2 bits per sample for the complex pixels. A complex pixel would then need an extra 2×4=8 bits. However, since each sample just needs 3 combinations, this is actually a bit of waste. A more compact way to store these bits per pixel is described next. The number of combinations needed is 3×3×3×3=81, and hence it suffices with 7 bits (instead of 8 bits), i.e., each sample needs three combinations to point into the palette.

In one embodiment, 64 bits may be used to store medium and complex pixels. If that number of bits is insufficient for the current tile, then compression succeeds. If not, compression fails and another technique is used or the data may be sent uncompressed.

Alternatively, color 2 in the table above can be the clear color, and in that case, color 2 does not have to be stored, which saves 32 bits, and these saved bits can be used to store more medium and complex pixels.

In accordance with another embodiment, for 8× MSAA, for a tile size of 8×4 pixels, there are 3 index bits per sample and 8*4*24=768 bits for all index bits for a tile. So for 8× MSAA with 4 bits per pixel to indicate status, sums to 4*8*4=128 bits. A 6-color palette costs 6*32=192 bits. The four bits per pixel can be chosen in any way, as long as one combination is left for the case where the samples in the pixel can point to all colors in the palette.

So where the palette has 2^n colors and each sample uses n palette index bits to point into the palette, the number of palette index bits for all samples in the tile is less than or equal to the sum of the index bits of the tile.

One embodiment may be as follows:

0000=the samples in the pixel only use color 0, and hence, nothing more needs to be stored for that pixel
0001=same but for color 1 in the palette
0010=same but for color 2 in the palette
0011=same but for color 3 in the palette
0100=same but for color 4 in the palette
0101=same but for color 5 in the palette
0110=the samples in the pixel only refer to color 0 or color 1, which means that each sample needs a 1 palette index bit, so these bits are appended to the compressed output stream (i.e., into the 768 index bits of the tile).
0111=same as above, but refer to color 0 and color 2
1000=same as above, but refer to color 0 and color 3
1001=same as above, but refer to color 0 and color 4
1010=same as above, but refer to color 0 and color 5
1011=same as above, but refer to color 1 and color 2
1100=same as above, but refer to color 1 and color 3
1101=same as above, but refer to color 1 and color 3
1110=same as above, but refer to color 1 and color 4
1111=complex pixel, a 3 bit index to the color palette per pixel is stored (or 6*6*6*6=1296 but that requires 11 bits for the first 4 palette indices, and the next 4 samples in the pixel can be stored using 11 bits as well, so this amounts to 11+11 bits, which is less than 3*8=24 bits. The 24-bit solution is simpler in hardware, but the 22 bit solution is more efficient in storage).

Another example is to use an 8-color palette for 8× MSAA, and still use 4 status bits per pixel, with the same table above, except that the last complex pixel is changed to 1111=complex pixel, where each sample in a complex pixel stores 3 palette index bits. Then the palette has 2^n=8 colors but each sample uses less than n palette index bits to point into the palette.

In accordance with one embodiment, a sequence **10**, shown in FIG. **2** may be implemented in software, firmware and/or hardware. In software and firmware embodiments it may be implemented by computer executed instructions stored in one or more non-transitory computer readable media such as magnetic, optical or semiconductor storage. In one embodiment, the storage may be part of a graphics processor.

The sequence begins by selecting a tile for eviction from the color buffer cache as indicated in block **12**. The index bits are investigated to see how many planes are used as indicated in block **14**. Then a check at diamond **16** determines if all the planes are in the cache.

If so, the tile is attempted to be compressed using a palette of three colors as one example as indicated in block **18**. Then a check at diamond **20** determines whether the desired bit budget is met by the attempted compression. If so, an indication of the successful compression within the budget is stored in the control surface as indicated in block **22**. Then the compressed representation is stored in the index bits as indicated in block **24**. Finally, the corresponding cache lines are marked (block **26**) as evicted. The evicted bits can be used for anything now. However, nothing is written back to memory because the compression representation is in the index bits.

If however, all planes are already in the cache, then another compression method must be used as indicated in block **28**.

If the bit budget is not met as determined in diamond **20**, then one can proceed at step 2 in the alternate compression method described next (block **30**).

An alternate compression method works as follows:

1. A cache line in the color buffer cache is selected for eviction—possibly more than one cache lines may be selected for eviction, i.e., both the cache line storing the left 4×4 pixels and the right 4×4 pixels in an 8×4 tile.

2. The 8×4 color plane is attempted to be compressed using some algorithm, including the palette-based compression method(s) presented above.

3. If a palette-based compression method succeeds, then this is indicated in the control surface bits of the tile, and the compressed representation is stored in the index bits. Else if another compression methods succeeds, then the data is sent in compressed form over the bus. This is indicated in the control surface bits of the tile.

4. Otherwise, it is sent in uncompressed form. This is also indicated in the control surface bits of the tile.

For decompression:

1. First check the control bits if the tile is marked as using a palette compressed representation in the index bits (FIG. **1**, block **22**). If yes, do not read anything from memory, simply decompress the data using the index bits, and populate the color cache.

2. Otherwise, continue as usual.

The example above was given with respect to 4× MSAA, i.e., four samples per pixel. However, it is straightforward to extend to other rates as well, e.g., 8× MSAA, 16× MSAA, etc., and even CSAA-like schemes.

For example, with 8× MSAA, each sample stores 3 index bits, and each pixel stores 24 (3*8) index bits. With 8×4 tiles, this amounts to 768 bits. Then 128 (32*4) bits are used for 4 colors with 32 bits each. There are 256 (8*8*4) samples in one tile. Then 2 bits per sample are stored: 256*2=512 and 512+128=640 bits. So, in this case, one can easily store a 4-color palette. Method B can also be augmented to 8× MSAA, in which case a 6 color-palette may be possible in the same way that a 3-color palette was possible in method B for 4× MSAA.

Tiles that are relatively simple can be compressed with a very small number of bits, and this scheme can trigger a lot in user interfaces, for example. Saving memory traffic is extremely important.

FIG. **3** depicts a graphics rendering system **50** according to one embodiment. A renderer **52** supplies data to two compressor/decompressor units **54** and **56**. The unit **54** may implement the algorithm **62**, also shown in FIG. **2**, and the unit **56** may implement the algorithm **64** described above as the alternate algorithm. Both compressors then interface with the memory interface **58** which in turn is communicating with the graphics memory **60**. The algorithm may be implemented in the color compressor unit **62** that handles MSAA.

The color cache **66** may be between the renderer **52** and the compressor/decompressors **54** and **56** (in which case the data is decompressed before it gets inserted into the cache, for example), or the cache **68** (indicated in dashed lines) between the compressor/decompressors and the memory **60**, and in this case the data is stored in compressed form in the cache.

FIG. **4** is a block diagram of a data processing system **100**, according to an embodiment. The data processing system **100** includes one or more processors **102** and one or more graphics processors **108**, and may be a single processor desktop system, a multiprocessor workstation system, or a server system having a large number of processors **102** or processor cores **107**. In on embodiment, the data processing system **100** is a system on a chip integrated circuit (SOC) for use in mobile, handheld, or embedded devices.

An embodiment of the data processing system **100** can include, or be incorporated within a server-based gaming platform, a game console, including a game and media console, a mobile gaming console, a handheld game console, or an online game console. In one embodiment, the data processing system **100** is a mobile phone, smart phone, tablet computing device or mobile Internet device. The data processing system **100** can also include, couple with, or be integrated within a wearable device, such as a smart watch wearable device, smart eyewear device, augmented reality device, or virtual reality device. In one embodiment, the data processing system **100** is a television or set top box device having one or more processors **102** and a graphical interface generated by one or more graphics processors **108**.

The one or more processors **102** each include one or more processor cores **107** to process instructions which, when executed, perform operations for system and user software. In one embodiment, each of the one or more processor cores **107** is configured to process a specific instruction set **109**. The instruction set **109** may facilitate complex instruction set computing (CISC), reduced instruction set computing (RISC), or computing via a very long instruction word (VLIW). Multiple processor cores **107** may each process a different instruction set **109** which may include instructions to facilitate the emulation of other instruction sets. A processor core **107** may also include other processing devices, such a digital signal processor (DSP).

In one embodiment, the processor **102** includes cache memory **104**. Depending on the architecture, the processor **102** can have a single internal cache or multiple levels of internal cache. In one embodiment, the cache memory is shared among various components of the processor **102**. In one embodiment, the processor **102** also uses an external cache (e.g., a Level 3 (L3) cache or last level cache (LLC)) (not shown) which may be shared among the processor cores **107** using known cache coherency techniques. A register file **106** is additionally included in the processor **102** which may include different types of registers for storing different types of data (e.g., integer registers, floating point registers, status registers, and an instruction pointer register). Some registers may be general-purpose registers, while other registers may be specific to the design of the processor **102**.

The processor **102** is coupled to a processor bus **110** to transmit data signals between the processor **102** and other components in the system **100**. The system **100** uses an exemplary 'hub' system architecture, including a memory controller hub **116** and an input output (I/O) controller hub **130**. The memory controller hub **116** facilitates communication between a memory device and other components of the system **100**, while the I/O controller hub (ICH) **130** provides connections to I/O devices via a local I/O bus.

The memory device **120**, can be a dynamic random access memory (DRAM) device, a static random access memory (SRAM) device, flash memory device, or some other memory device having suitable performance to serve as process memory. The memory **120** can store data **122** and instructions **121** for use when the processor **102** executes a process. The memory controller hub **116** also couples with an optional external graphics processor **112**, which may communicate with the one or more graphics processors **108** in the processors **102** to perform graphics and media operations.

The ICH **130** enables peripherals to connect to the memory **120** and processor **102** via a high-speed I/O bus. The I/O peripherals include an audio controller **146**, a firmware interface **128**, a wireless transceiver **126** (e.g., Wi-Fi, Bluetooth), a data storage device **124** (e.g., hard disk drive, flash memory, etc.), and a legacy I/O controller for coupling legacy (e.g., Personal System 2 (PS/2)) devices to the system. One or more Universal Serial Bus (USB) controllers **142** connect input devices, such as keyboard and mouse **144** combinations. A network controller **134** may also couple to the ICH **130**. In one embodiment, a high-performance network controller (not shown) couples to the processor bus **110**.

FIG. **5** is a block diagram of an embodiment of a processor **200** having one or more processor cores **202**A-N, an integrated memory controller **214**, and an integrated graphics processor **208**. The processor **200** can include additional cores up to and including additional core **202**N represented by the dashed lined boxes. Each of the cores **202**A-N includes one or more internal cache units **204**A-N. In one embodiment each core also has access to one or more shared cached units **206**.

The internal cache units **204**A-N and shared cache units **206** represent a cache memory hierarchy within the processor **200**. The cache memory hierarchy may include at least one level of instruction and data cache within each core and one or more levels of shared mid-level cache, such as a level 2 (L2), level 3 (L3), level 4 (L4), or other levels of cache, where the highest level of cache before external memory is classified as the last level cache (LLC). In one embodiment, cache coherency logic maintains coherency between the various cache units **206** and **204**A-N.

The processor **200** may also include a set of one or more bus controller units **216** and a system agent **210**. The one or more bus controller units manage a set of peripheral buses, such as one or more Peripheral Component Interconnect buses (e.g., PCI, PCI Express). The system agent **210** provides management functionality for the various processor components. In one embodiment, the system agent **210** includes one or more integrated memory controllers **214** to manage access to various external memory devices (not shown).

In one embodiment, one or more of the cores **202**A-N include support for simultaneous multi-threading. In such embodiment, the system agent **210** includes components for coordinating and operating cores **202**A-N during multi-threaded processing. The system agent **210** may additionally include a power control unit (PCU), which includes logic and components to regulate the power state of the cores **202**A-N and the graphics processor **208**.

The processor **200** additionally includes a graphics processor **208** to execute graphics processing operations. In one embodiment, the graphics processor **208** couples with the set of shared cache units **206**, and the system agent unit **210**, including the one or more integrated memory controllers **214**. In one embodiment, a display controller **211** is coupled with the graphics processor **208** to drive graphics processor output to one or more coupled displays. The display controller **211** may be separate module coupled with the graphics processor via at least one interconnect, or may be integrated within the graphics processor **208** or system agent **210**.

In one embodiment a ring based interconnect unit **212** is used to couple the internal components of the processor **200**, however an alternative interconnect unit may be used, such as a point to point interconnect, a switched interconnect, or other techniques, including techniques well known in the art. In one embodiment, the graphics processor **208** couples with the ring interconnect **212** via an I/O link **213**.

The exemplary I/O link **213** represents at least one of multiple varieties of I/O interconnects, including an on package I/O interconnect which facilitates communication between various processor components and a high-performance embedded memory module **218**, such as an eDRAM module. In one embodiment each of the cores **202**-N and the graphics processor **208** use the embedded memory modules **218** as shared last level cache.

In one embodiment cores **202**A-N are homogenous cores executing the same instruction set architecture. In another embodiment, the cores **202**A-N are heterogeneous in terms of instruction set architecture (ISA), where one or more of the cores **202**A-N execute a first instruction set, while at least one of the other cores executes a subset of the first instruction set or a different instruction set.

The processor **200** can be a part of or implemented on one or more substrates using any of a number of process technologies, for example, Complementary metal-oxide-semiconductor (CMOS), Bipolar Junction/Complementary metal-oxide-semiconductor (BiCMOS) or N-type metal-oxide-semiconductor logic (NMOS). Additionally, the processor **200** can be implemented on one or more chips or as a system on a chip (SOC) integrated circuit having the illustrated components, in addition to other components.

FIG. **6** is a block diagram of one embodiment of a graphics processor **300** which may be a discreet graphics processing unit, or may be graphics processor integrated with a plurality of processing cores. In one embodiment, the graphics processor is communicated with via a memory mapped I/O interface to registers on the graphics processor and via commands placed into the processor memory. The graphics processor **300** includes a memory interface **314** to access memory. The memory interface **314** can be an interface to local memory, one or more internal caches, one or more shared external caches, and/or to system memory.

The graphics processor **300** also includes a display controller **302** to drive display output data to a display device **320**. The display controller **302** includes hardware for one or more overlay planes for the display and composition of multiple layers of video or user interface elements. In one embodiment the graphics processor **300** includes a video codec engine **306** to encode, decode, or transcode media to, from, or between one or more media encoding formats, including, but not limited to Moving Picture Experts Group (MPEG) formats such as MPEG-2, Advanced Video Coding (AVC) formats such as H.264/MPEG-4 AVC, as well as the Society of Motion Picture & Television Engineers (SMPTE) 421M/VC-1, and Joint Photographic Experts Group (JPEG) formats such as JPEG, and Motion JPEG (MJPEG) formats.

In one embodiment, the graphics processor **300** includes a block image transfer (BLIT) engine **304** to perform two-dimensional (2D) rasterizer operations including, for example, bit-boundary block transfers. However, in one embodiment, 2D graphics operations are performed using one or more components of the graphics-processing engine (GPE) **310**. The graphics-processing engine **310** is a compute engine for performing graphics operations, including three-dimensional (3D) graphics operations and media operations.

The GPE **310** includes a 3D pipeline **312** for performing 3D operations, such as rendering three-dimensional images and scenes using processing functions that act upon 3D primitive shapes (e.g., rectangle, triangle, etc.). The 3D pipeline **312** includes programmable and fixed function elements that perform various tasks within the element and/or spawn execution threads to a 3D/Media sub-system **315**. While the 3D pipeline **312** can be used to perform media operations, an embodiment of the GPE **310** also includes a media pipeline **316** that is specifically used to perform media operations, such as video post processing and image enhancement.

In one embodiment, the media pipeline **316** includes fixed function or programmable logic units to perform one or more specialized media operations, such as video decode acceleration, video de-interlacing, and video encode acceleration in place of, or on behalf of the video codec engine **306**. In on embodiment, the media pipeline **316** additionally includes a thread spawning unit to spawn threads for execution on the 3D/Media sub-system **315**. The spawned threads perform computations for the media operations on one or more graphics execution units included in the 3D/Media sub-system.

The 3D/Media subsystem **315** includes logic for executing threads spawned by the 3D pipeline **312** and media pipeline **316**. In one embodiment, the pipelines send thread execution requests to the 3D/Media subsystem **315**, which includes thread dispatch logic for arbitrating and dispatching the various requests to available thread execution resources. The execution resources include an array of graphics execution units to process the 3D and media threads. In one embodiment, the 3D/Media subsystem **315** includes one or more internal caches for thread instructions and data. In one embodiment, the subsystem also includes shared memory, including registers and addressable memory, to share data between threads and to store output data.

FIG. **7** is a block diagram of an embodiment of a graphics processing engine **410** for a graphics processor. In one embodiment, the graphics processing engine (GPE) **410** is a version of the GPE **310** shown in FIG. **5**. The GPE **410** includes a 3D pipeline **412** and a media pipeline **416**, each of which can be either different from or similar to the implementations of the 3D pipeline **312** and the media pipeline **316** of FIG. **5**.

In one embodiment, the GPE **410** couples with a command streamer **403**, which provides a command stream to the GPE 3D and media pipelines **412**, **416**. The command streamer **403** is coupled to memory, which can be system memory, or one or more of internal cache memory and shared cache memory. The command streamer **403** receives commands from the memory and sends the commands to the 3D pipeline **412** and/or media pipeline **416**. The 3D and media pipelines process the commands by performing operations via logic within the respective pipelines or by dispatching one or more execution threads to the execution unit array **414**. In one embodiment, the execution unit array **414** is scalable, such that the array includes a variable number of execution units based on the target power and performance level of the GPE **410**.

A sampling engine **430** couples with memory (e.g., cache memory or system memory) and the execution unit array **414**. In one embodiment, the sampling engine **430** provides a memory access mechanism for the scalable execution unit array **414** that allows the execution array **414** to read graphics and media data from memory. In one embodiment, the sampling engine **430** includes logic to perform specialized image sampling operations for media.

The specialized media sampling logic in the sampling engine **430** includes a de-noise/de-interlace module **432**, a motion estimation module **434**, and an image scaling and filtering module **436**. The de-noise/de-interlace module **432** includes logic to perform one or more of a de-noise or a de-interlace algorithm on decoded video data. The de-interlace logic combines alternating fields of interlaced video content into a single fame of video. The de-noise logic reduces or remove data noise from video and image data. In one embodiment, the de-noise logic and de-interlace logic are motion adaptive and use spatial or temporal filtering based on the amount of motion detected in the video data. In one embodiment, the de-noise/de-interlace module **432** includes dedicated motion detection logic (e.g., within the motion estimation engine **434**).

The motion estimation engine **434** provides hardware acceleration for video operations by performing video acceleration functions such as motion vector estimation and prediction on video data. The motion estimation engine determines motion vectors that describe the transformation of image data between successive video frames. In one embodiment, a graphics processor media codec uses the video motion estimation engine **434** to perform operations on video at the macro-block level that may otherwise be computationally intensive to perform using a general-purpose processor. In one embodiment, the motion estimation engine **434** is generally available to graphics processor components to assist with video decode and processing functions that are sensitive or adaptive to the direction or magnitude of the motion within video data.

The image scaling and filtering module **436** performs image-processing operations to enhance the visual quality of generated images and video. In one embodiment, the scaling and filtering module **436** processes image and video data during the sampling operation before providing the data to the execution unit array **414**.

In one embodiment, the graphics processing engine **410** includes a data port **444**, which provides an additional mechanism for graphics subsystems to access memory. The data port **444** facilitates memory access for operations including render target writes, constant buffer reads, scratch memory space reads/writes, and media surface accesses. In one embodiment, the data port **444** includes cache memory space to cache accesses to memory. The cache memory can be a single data cache or separated into multiple caches for the multiple subsystems that access memory via the data port (e.g., a render buffer cache, a constant buffer cache, etc.). In one embodiment, threads executing on an execution unit in the execution unit array **414** communicate with the data port by exchanging messages via a data distribution interconnect that couples each of the sub-systems of the graphics processing engine **410**.

FIG. **8** is a block diagram of another embodiment of a graphics processor. In one embodiment, the graphics processor includes a ring interconnect **502**, a pipeline front-end **504**, a media engine **537**, and graphics cores **580**A-N. The ring interconnect **502** couples the graphics processor to other processing units, including other graphics processors or one or more general-purpose processor cores. In one embodiment, the graphics processor is one of many processors integrated within a multi-core processing system.

The graphics processor receives batches of commands via the ring interconnect **502**. The incoming commands are interpreted by a command streamer **503** in the pipeline front-end **504**. The graphics processor includes scalable execution logic to perform 3D geometry processing and media processing via the graphics core(s) **580**A-N. For 3D geometry processing commands, the command streamer **503** supplies the commands to the geometry pipeline **536**. For at least some media processing commands, the command streamer **503** supplies the commands to a video front end **534**, which couples with a media engine **537**. The media engine **537** includes a video quality engine (VQE) **530** for video and image post processing and a multi-format encode/decode (MFX) **533** engine to provide hardware-accelerated media data encode and decode. The geometry pipeline **536** and media engine **537** each generate execution threads for the thread execution resources provided by at least one graphics core **580**A.

The graphics processor includes scalable thread execution resources featuring modular cores **580**A-N (sometime referred to as core slices), each having multiple sub-cores **550**A-N, **560**A-N (sometimes referred to as core sub-slices). The graphics processor can have any number of graphics cores **580**A through **580**N. In one embodiment, the graphics processor includes a graphics core **580**A having at least a first sub-core **550**A and a second core sub-core **560**A. In another embodiment, the graphics processor is a low power processor with a single sub-core (e.g., **550**A). In one embodiment, the graphics processor includes multiple graphics cores **580**A-N, each including a set of first sub-cores **550**A-N and a set of second sub-cores **560**A-N. Each sub-core in the set of first sub-cores **550**A-N includes at least a first set of execution units **552**A-N and media/texture samplers **554**A-N. Each sub-core in the set of second sub-cores **560**A-N includes at least a second set of execution units **562**A-N and samplers **564**A-N. In one embodiment, each sub-core **550**A-N, **560**A-N shares a set of shared resources **570**A-N. In one embodiment, the shared resources include shared cache memory and pixel operation logic. Other shared resources may also be included in the various embodiments of the graphics processor.

FIG. **9** illustrates thread execution logic **600** including an array of processing elements employed in one embodiment of a graphics processing engine. In one embodiment, the thread execution logic **600** includes a pixel shader **602**, a thread dispatcher **604**, instruction cache **606**, a scalable execution unit array including a plurality of execution units **608**A-N, a sampler **610**, a data cache **612**, and a data port **614**. In one embodiment the included components are interconnected via an interconnect fabric that links to each of the components. The thread execution logic **600** includes one or more connections to memory, such as system memory or cache memory, through one or more of the instruction cache **606**, the data port **614**, the sampler **610**, and the execution unit array **608**A-N. In one embodiment, each execution unit (e.g. **608**A) is an individual vector processor capable of executing multiple simultaneous threads and processing multiple data elements in parallel for each thread. The execution unit array **608**A-N includes any number individual execution units.

In one embodiment, the execution unit array **608**A-N is primarily used to execute "shader" programs. In one embodiment, the execution units in the array **608**A-N execute an instruction set that includes native support for many standard 3D graphics shader instructions, such that shader programs from graphics libraries (e.g., Direct 3D and OpenGL) are executed with a minimal translation. The execution units support vertex and geometry processing (e.g., vertex programs, geometry programs, vertex shaders), pixel processing (e.g., pixel shaders, fragment shaders) and general-purpose processing (e.g., compute and media shaders).

Each execution unit in the execution unit array **608**A-N operates on arrays of data elements. The number of data elements is the "execution size," or the number of channels for the instruction. An execution channel is a logical unit of execution for data element access, masking, and flow control within instructions. The number of channels may be independent of the number of physical ALUs or FPUs for a particular graphics processor. The execution units **608**A-N support integer and floating-point data types.

The execution unit instruction set includes single instruction multiple data (SIMD) instructions. The various data elements can be stored as a packed data type in a register and the execution unit will process the various elements based on the data size of the elements. For example, when operating on a 256-bit wide vector, the 256 bits of the vector are stored in a register and the execution unit operates on the vector as four separate 64-bit packed data elements (quad-word (QW) size data elements), eight separate 32-bit packed data elements (double word (DW) size data elements), sixteen separate 16-bit packed data elements (word (W) size data elements), or thirty-two separate 8-bit data elements (byte (B) size data elements). However, different vector widths and register sizes are possible.

One or more internal instruction caches (e.g., **606**) are included in the thread execution logic **600** to cache thread instructions for the execution units. In one embodiment, one or more data caches (e.g., **612**) are included to cache thread data during thread execution. A sampler **610** is included to provide texture sampling for 3D operations and media sampling for media operations. In one embodiment, the sampler **610** includes specialized texture or media sampling functionality to process texture or media data during the sampling process before providing the sampled data to an execution unit.

During execution, the graphics and media pipelines send thread initiation requests to the thread execution logic **600** via thread spawning and dispatch logic. The thread execution logic **600** includes a local thread dispatcher **604** that arbitrates thread initiation requests from the graphics and media pipelines and instantiates the requested threads on one or more execution units **608**A-N. For example, the geometry pipeline (e.g., **536** of FIG. **6**) dispatches vertex processing, tessellation, or geometry processing threads to the thread execution logic **600**. The thread dispatcher **604** can also process runtime thread spawning requests from the executing shader programs.

Once a group of geometric objects have been processed and rasterized into pixel data, the pixel shader **602** is invoked to further compute output information and cause results to be written to output surfaces (e.g., color buffers, depth buffers, stencil buffers, etc.). In one embodiment, the pixel shader **602** calculates the values of the various vertex attributes that are to be interpolated across the rasterized object. The pixel shader **602** then executes an API-supplied pixel shader program. To execute the pixel shader program, the pixel shader **602** dispatches threads to an execution unit (e.g., **608**A) via the thread dispatcher **604**. The pixel shader **602** uses texture sampling logic in the sampler **610** to access texture data in texture maps stored in memory. Arithmetic operations on the texture data and the input geometry data compute pixel color data for each geometric fragment, or discards one or more pixels from further processing.

In one embodiment, the data port **614** provides a memory access mechanism for the thread execution logic **600** output processed data to memory for processing on a graphics processor output pipeline. In one embodiment, the data port **614** includes or couples to one or more cache memories (e.g., data cache **612**) to cache data for memory access via the data port.

FIG. **10** is a block diagram illustrating a graphics processor execution unit instruction format according to an embodiment. In one embodiment, the graphics processor execution units support an instruction set having instructions in multiple formats. The solid lined boxes illustrate the components that are generally included in an execution unit instruction, while the dashed lines include components that are optional or that are only included in a sub-set of the instructions. The instruction format described an illustrated are macro-instructions, in that they are instructions supplied to the execution unit, as opposed to micro-operations resulting from instruction decode once the instruction is processed.

In one embodiment, the graphics processor execution units natively support instructions in a 128-bit format **710**. A 64-bit compacted instruction format **730** is available for some instructions based on the selected instruction, instruction options, and number of operands. The native 128-bit format **710** provides access to all instruction options, while some options and operations are restricted in the 64-bit format **730**. The native instructions available in the 64-bit format **730** varies by embodiment. In one embodiment, the instruction is compacted in part using a set of index values in an index field **713**. The execution unit hardware references a set of compaction tables based on the index values and uses the compaction table outputs to reconstruct a native instruction in the 128-bit format **710**.

For each format, an instruction opcode **712** defines the operation that the execution unit is to perform. The execution units execute each instruction in parallel across the multiple data elements of each operand. For example, in response to an add instruction the execution unit performs a simultaneous add operation across each color channel representing a texture element or picture element. By default, the execution unit performs each instruction across all data channels of the operands. An instruction control field **712** enables control over certain execution options, such as channels selection (e.g., predication) and data channel order (e.g., swizzle). For 128-bit instructions **710** an exec-size field **716** limits the number of data channels that will be executed in parallel. The exec-size field **716** is not available for use in the 64-bit compact instruction format **730**.

Some execution unit instructions have up to three operands including two source operands, src0 **720**, src1 **722**, and one destination **718**. In one embodiment, the execution units support dual destination instructions, where one of the destinations is implied. Data manipulation instructions can have a third source operand (e.g., SRC2 **724**), where the instruction opcode JJ12 determines the number of source operands. An instruction's last source operand can be an immediate (e.g., hard-coded) value passed with the instruction.

In one embodiment instructions are grouped based on opcode bit-fields to simplify Opcode decode **740**. For an 8-bit opcode, bits **4**, **5**, and **6** allow the execution unit to determine the type of opcode. The precise opcode grouping shown is exemplary. In one embodiment, a move and logic opcode group **742** includes data movement and logic instructions (e.g., mov, cmp). The move and logic group **742** shares the five most significant bits (MSB), where move instructions are in the form of 0000xxxxb (e.g., 0x0x) and logic instructions are in the form of 0001xxxxb (e.g., 0x01). A flow control instruction group **744** (e.g., call, jmp) includes instructions in the form of 0010xxxxb (e.g., 0x20). A miscellaneous instruction group **746** includes a mix of instructions, including synchronization instructions (e.g., wait, send) in the form of 0011xxxxb (e.g., 0x30). A parallel math instruction group **748** includes component-wise arithmetic instructions (e.g., add, mul) in the form of 0100xxxxb (e.g., 0x40). The parallel math group **748** performs the arithmetic operations in parallel across data channels. The vector math group **750** includes arithmetic instructions (e.g., dp4) in the form of 0101xxxxb (e.g., 0x50). The vector math group performs arithmetic such as dot product calculations on vector operands.

FIG. **11** is a block diagram of another embodiment of a graphics processor which includes a graphics pipeline **820**, a media pipeline **830**, a display engine **840**, thread execution logic **850**, and a render output pipeline **870**. In one embodiment, the graphics processor is a graphics processor within a multi-core processing system that includes one or more general purpose processing cores. The graphics processor is controlled by register writes to one or more control registers (not shown) or via commands issued to the graphics processor via a ring interconnect **802**. The ring interconnect **802** couples the graphics processor to other processing components, such as other graphics processors or general-purpose processors. Commands from the ring interconnect are interpreted by a command streamer **803** which supplies instructions to individual components of the graphics pipeline **820** or media pipeline **830**.

The command streamer **803** directs the operation of a vertex fetcher **805** component that reads vertex data from memory and executes vertex-processing commands provided by the command streamer **803**. The vertex fetcher **805** provides vertex data to a vertex shader **807**, which performs coordinate space transformation and lighting operations to each vertex. The vertex fetcher **805** and vertex shader **807** execute vertex-processing instructions by dispatching execution threads to the execution units **852**A, **852**B via a thread dispatcher **831**.

In one embodiment, the execution units **852**A, **852**B are an array of vector processors having an instruction set for performing graphics and media operations. The execution units **852**A, **852**B have an attached L1 cache **851** that is specific for each array or shared between the arrays. The cache can be configured as a data cache, an instruction cache, or a single cache that is partitioned to contain data and instructions in different partitions.

In one embodiment, the graphics pipeline **820** includes tessellation components to perform hardware-accelerated tessellation of 3D objects. A programmable hull shader **811** configures the tessellation operations. A programmable domain shader **817** provides back-end evaluation of tessellation output. A tessellator **813** operates at the direction of the hull shader **811** and contains special purpose logic to generate a set of detailed geometric objects based on a coarse geometric model that is provided as input to the graphics pipeline **820**. If tessellation is not used, the tessellation components **811**, **813**, **817** can be bypassed.

The complete geometric objects can be processed by a geometry shader **819** via one or more threads dispatched to the execution units **852**A, **852**B, or can proceed directly to the clipper **829**. The geometry shader operates on entire geometric objects, rather than vertices or patches of vertices as in previous stages of the graphics pipeline. If the tessellation is disabled the geometry shader **819** receives input from the vertex shader **807**. The geometry shader **819** is programmable by a geometry shader program to perform geometry tessellation if the tessellation units are disabled.

Prior to rasterization, vertex data is processed by a clipper **829**, which is either a fixed function clipper or a programmable clipper having clipping and geometry shader functions. In one embodiment, a rasterizer **873** in the render output pipeline **870** dispatches pixel shaders to convert the geometric objects into their per pixel representations. In one embodiment, pixel shader logic is included in the thread execution logic **850**.

The graphics engine has an interconnect bus, interconnect fabric, or some other interconnect mechanism that allows data and message passing amongst the major components of the graphics engine. In one embodiment the execution units **852**A, **852**B and associated cache(s) **851**, texture and media sampler **854**, and texture/sampler cache **858** interconnect via a data port **856** to perform memory access and communicate with render output pipeline components of the graphics engine. In one embodiment, the sampler **854**, caches **851**, **858** and execution units **852**A, **852**B each have separate memory access paths.

In one embodiment, the render output pipeline **870** contains a rasterizer and depth test component **873** that converts vertex-based objects into their associated pixel-based representation. In one embodiment, the rasterizer logic includes a windower/masker unit to perform fixed function triangle and line rasterization. An associated render and depth buffer caches **878**, **879** are also available in one embodiment. A pixel operations component **877** performs pixel-based operations on the data, though in some instances, pixel operations associated with 2D operations (e.g. bit block image transfers with blending) are performed by the 2D engine **841**, or substituted at display time by the display controller **843** using overlay display planes. In one embodiment a shared L3 cache **875** is available to all graphics components, allowing the sharing of data without the use of main system memory.

The graphics processor media pipeline **830** includes a media engine **837** and a video front end **834**. In one embodiment, the video front end **834** receives pipeline commands from the command streamer **803**. However, in one embodiment the media pipeline **830** includes a separate command streamer. The video front-end **834** processes media commands before sending the command to the media engine **837**. In one embodiment, the media engine includes thread spawning functionality to spawn threads for dispatch to the thread execution logic **850** via the thread dispatcher **831**.

In one embodiment, the graphics engine includes a display engine **840**. In one embodiment, the display engine **840** is external to the graphics processor and couples with the graphics processor via the ring interconnect **802**, or some other interconnect bus or fabric. The display engine **840** includes a 2D engine **841** and a display controller **843**. The display engine **840** contains special purpose logic capable of operating independently of the 3D pipeline. The display controller **843** couples with a display device (not shown), which may be a system integrated display device, as in a laptop computer, or an external display device attached via an display device connector.

The graphics pipeline **820** and media pipeline **830** are configurable to perform operations based on multiple graphics and media programming interfaces and are not specific to any one application programming interface (API). In one embodiment, driver software for the graphics processor translates API calls that are specific to a particular graphics or media library into commands that can be processed by the graphics processor. In various embodiments, support is provided for the Open Graphics Library (OpenGL) and Open Computing Language (OpenCL) supported by the Khronos Group, the Direct3D library from the Microsoft Corporation, or, in one embodiment, both OpenGL and D3D. Support may also be provided for the Open Source Computer Vision Library (OpenCV). A future API with a compatible 3D pipeline would also be supported if a mapping can be made from the pipeline of the future API to the pipeline of the graphics processor.

FIG. **12**A is a block diagram illustrating a graphics processor command format according to an embodiment and FIG. **12**B is a block diagram illustrating a graphics processor command sequence according to an embodiment. The solid lined boxes in FIG. **12**A illustrate the components that are generally included in a graphics command while the dashed lines include components that are optional or that are only included in a sub-set of the graphics commands. The exemplary graphics processor command format **900** of FIG. **12**A includes data fields to identify a target client **902** of the command, a command operation code (opcode) **904**, and the relevant data **906** for the command. A sub-opcode **905** and a command size **908** are also included in some commands.

The client **902** specifies the client unit of the graphics device that processes the command data. In one embodiment, a graphics processor command parser examines the client field of each command to condition the further processing of the command and route the command data to the appropriate client unit. In one embodiment, the graphics processor client units include a memory interface unit, a render unit, a 2D unit, a 3D unit, and a media unit. Each client unit has a corresponding processing pipeline that processes the commands. Once the command is received by the client unit, the client unit reads the opcode **904** and, if present, sub-opcode **905** to determine the operation to perform. The client unit performs the command using information in the data **906** field of the command. For some commands an explicit command size **908** is expected to specify the size of the command. In one embodiment, the command parser automatically determines the size of at least some of the commands based on the command opcode. In one embodiment commands are aligned via multiples of a double word.

The flow chart in FIG. **12**B shows a sample command sequence **910**. In one embodiment, software or firmware of a data processing system that features an embodiment of the graphics processor uses a version of the command sequence shown to set up, execute, and terminate a set of graphics operations. A sample command sequence is shown and described for exemplary purposes, however embodiments are not limited to these commands or to this command sequence. Moreover, the commands may be issued as batch of commands in a command sequence, such that the graphics processor will process the sequence of commands in an at least partially concurrent manner.

The sample command sequence **910** may begin with a pipeline flush command **912** to cause any active graphics pipeline to complete the currently pending commands for the pipeline. In one embodiment, the 3D pipeline **922** and the media pipeline **924** do not operate concurrently. The pipeline flush is performed to cause the active graphics pipeline to complete any pending commands. In response to a pipeline flush, the command parser for the graphics processor will pause command processing until the active drawing engines complete pending operations and the relevant read caches are invalidated. Optionally, any data in the render cache that is marked 'dirty' can be flushed to memory. A pipeline flush command **912** can be used for pipeline synchronization or before placing the graphics processor into a low power state.

A pipeline select command **913** is used when a command sequence requires the graphics processor to explicitly switch between pipelines. A pipeline select command **913** is required only once within an execution context before issuing pipeline commands unless the context is to issue commands for both pipelines. In one embodiment, a pipeline flush command is **912** is required immediately before a pipeline switch via the pipeline select command **913**.

A pipeline control command **914** configures a graphics pipeline for operation and is used to program the 3D pipeline **922** and the media pipeline **924**. The pipeline control command **914** configures the pipeline state for the active pipeline. In one embodiment, the pipeline control command **914** is used for pipeline synchronization and to clear data from one or more cache memories within the active pipeline before processing a batch of commands.

Return buffer state commands **916** are used to configure a set of return buffers for the respective pipelines to write data. Some pipeline operations require the allocation, selection, or configuration of one or more return buffers into which the operations write intermediate data during processing. The graphics processor also uses one or more return buffers to store output data and to perform cross thread communication. The return buffer state **916** includes selecting the size and number of return buffers to use for a set of pipeline operations.

The remaining commands in the command sequence differ based on the active pipeline for operations. Based on a pipeline determination **920**, the command sequence is tailored to the 3D pipeline **922** beginning with the 3D pipeline state **930**, or the media pipeline **924** beginning at the media pipeline state **940**.

The commands for the 3D pipeline state **930** include 3D state setting commands for vertex buffer state, vertex element state, constant color state, depth buffer state, and other state variables that are to be configured before 3D primitive commands are processed. The values of these commands are determined at least in part based the particular 3D API in use. 3D pipeline state **930** commands are also able to selectively disable or bypass certain pipeline elements if those elements will not be used.

The 3D primitive **932** command is used to submit 3D primitives to be processed by the 3D pipeline. Commands and associated parameters that are passed to the graphics processor via the 3D primitive **932** command are forwarded to the vertex fetch function in the graphics pipeline. The vertex fetch function uses the 3D primitive **932** command data to generate vertex data structures. The vertex data structures are stored in one or more return buffers. The 3D primitive **932** command is used to perform vertex operations on 3D primitives via vertex shaders. To process vertex shaders, the 3D pipeline **922** dispatches shader execution threads to graphics processor execution units.

The 3D pipeline **922** is triggered via an execute **934** command or event. In one embodiment a register write triggers command execution. In one embodiment execution is triggered via a 'go' or 'kick' command in the command sequence. In one embodiment command execution is triggered using a pipeline synchronization command to flush the command sequence through the graphics pipeline. The 3D pipeline will perform geometry processing for the 3D primitives. Once operations are complete, the resulting geometric objects are rasterized and the pixel engine colors the resulting pixels. Additional commands to control pixel shading and pixel back end operations may also be included for those operations.

The sample command sequence **910** follows the media pipeline **924** path when performing media operations. In general, the specific use and manner of programming for the media pipeline **924** depends on the media or compute operations to be performed. Specific media decode operations may be offloaded to the media pipeline during media decode. The media pipeline can also be bypassed and media decode can be performed in whole or in part using resources provided by one or more general purpose processing cores. In one embodiment, the media pipeline also includes elements for general-purpose graphics processor unit (GPGPU) operations, where the graphics processor is used to perform SIMD vector operations using computational shader programs that are not explicitly related to the rendering of graphics primitives.

The media pipeline **924** is configured in a similar manner as the 3D pipeline **922**. A set of media pipeline state commands **940** are dispatched or placed into in a command queue before the media object commands **942**. The media pipeline state commands **940** include data to configure the media pipeline elements that will be used to process the media objects. This includes data to configure the video decode and video encode logic within the media pipeline, such as encode or decode format. The media pipeline state commands **940** also support the use one or more pointers to "indirect" state elements that contain a batch of state settings.

Media object commands **942** supply pointers to media objects for processing by the media pipeline. The media objects include memory buffers containing video data to be processed. In one embodiment, all media pipeline state must be valid before issuing a media object command **942**. Once the pipeline state is configured and media object commands **942** are queued, the media pipeline **924** is triggered via an execute **934** command or an equivalent execute event (e.g., register write). Output from the media pipeline **924** may then be post processed by operations provided by the 3D pipeline **922** or the media pipeline **924**. In one embodiment, GPGPU operations are configured and executed in a similar manner as media operations.

FIG. **13** illustrates exemplary graphics software architecture for a data processing system according to an embodiment. The software architecture includes a 3D graphics application **1010**, an operating system **1020**, and at least one processor **1030**. The processor **1030** includes a graphics processor **1032** and one or more general-purpose processor core(s) **1034**. The graphics application **1010** and operating system **1020** each execute in the system memory **1050** of the data processing system.

In one embodiment, the 3D graphics application **1010** contains one or more shader programs including shader instructions **1012**. The shader language instructions may be in a high-level shader language, such as the High Level Shader Language (HLSL) or the OpenGL Shader Language (GLSL). The application also includes executable instructions **1014** in a machine language suitable for execution by the general-purpose processor core **1034**. The application also includes graphics objects **1016** defined by vertex data.

The operating system **1020** may be a Microsoft® Windows® operating system from the Microsoft Corporation, a proprietary UNIX-like operating system, or an open source UNIX-like operating system using a variant of the Linux kernel. When the Direct3D API is in use, the operating system **1020** uses a front-end shader compiler **1024** to compile any shader instructions **1012** in HLSL into a lower-level shader language. The compilation may be a just-in-time compilation or the application can perform share pre-compilation. In one embodiment, high-level shaders are compiled into low-level shaders during the compilation of the 3D graphics application **1010**.

The user mode graphics driver **1026** may contain a back-end shader compiler **1027** to convert the shader instructions **1012** into a hardware specific representation. When the OpenGL API is in use, shader instructions **1012** in the GLSL high-level language are passed to a user mode graphics driver **1026** for compilation. The user mode graphics driver uses operating system kernel mode functions **1028** to communicate with a kernel mode graphics driver **1029**. The kernel mode graphics driver **1029** communicates with the graphics processor **1032** to dispatch commands and instructions.

To the extent various operations or functions are described herein, they can be described or defined as hardware circuitry, software code, instructions, configuration, and/or data. The content can be embodied in hardware logic, or as directly executable software ("object" or "executable" form), source code, high level shader code designed for execution on a graphics engine, or low level assembly language code in an instruction set for a specific processor or graphics core. The software content of the embodiments described herein can be provided via an article of manufacture with the content stored thereon, or via a method of operating a communication interface to send data via the communication interface.

A non-transitory machine readable storage medium can cause a machine to perform the functions or operations described, and includes any mechanism that stores information in a form accessible by a machine (e.g., computing device, electronic system, etc.), such as recordable/non-recordable media (e.g., read only memory (ROM), random access memory (RAM), magnetic disk storage media, optical storage media, flash memory devices, etc.). A communication interface includes any mechanism that interfaces to any of a hardwired, wireless, optical, etc., medium to communicate to another device, such as a memory bus interface, a processor bus interface, an Internet connection, a disk controller, etc. The communication interface is configured by providing configuration parameters or sending signals to prepare the communication interface to provide a data signal describing the software content. The communication interface can be accessed via one or more commands or signals sent to the communication interface.

Various components described can be a means for performing the operations or functions described. Each component described herein includes software, hardware, or a combination of these. The components can be implemented as software modules, hardware modules, special-purpose hardware (e.g., application specific hardware, application specific integrated circuits (ASICs), digital signal processors (DSPs), etc.), embedded controllers, hardwired circuitry, etc. Besides what is described herein, various modifications can be made to the disclosed embodiments and implementations of the invention without departing from their scope. Therefore, the illustrations and examples herein should be construed in an illustrative, and not a restrictive sense. The scope of the invention should be measured solely by reference to the claims that follow.

The following clauses and/or examples pertain to further embodiments:

One example embodiment may be a method comprising attempting to compress cache lines using a color palette, checking whether the palette compression meets a bit budget, and if so, storing an indicator in a control surface that palette compressor was used, storing the compressed representation in index bits in a color buffer and marking the cache lines as evicted. The method may also include where the palette is a set of N colors, and each sample uses N combinations to point into the palette, where the sum of the storage of the palette and the sum of all samples' combinations can be stored in index bits of a tile. The method may also include where the palette is 2^n colors, and each sample uses n palette index bits to point into the palette, and where the sum of the storage of the palette and the number of palette index bits for all samples in the tile is less than or equal to the sum of the index bits of the tile. The method may also include wherein the palette is 2^n colors, and each sample uses less than n palette index bits to point into the palette. The method may also include selecting a tile, including cache lines, for eviction from a color buffer cache and checking index bits in the color cache to determine a number of planes in the color buffer and checking to determine whether all of said planes are in a color buffer cache. The method may also include only attempting to compress if all of the planes are in the color buffer cache. The method may also include using a different compression if all of the planes are not in the cache. The method may also include using a different compression if the bit budget is not met. The method may also include for 4× MSAA, storing a 2-color palette, and 1 palette index bit per sample. The method may also include for 4× MSAA, storing 3 status bits per pixel indicating how the palette indices are encoded. The method may also include for 8× MSAA, storing a 4-color palette, and 2 palette index bits per sample. The method may also include for 8× MSAA, storing 4 status bits per pixel indicating how the palette indices are encoded.

In another example embodiment may be one or more non-transitory computer readable media storing instructions to execute a sequence comprising attempting to compress cache lines using a color palette, checking whether the palette compression meets a bit budget, and if so, storing an indicator in a control surface that palette compressor was used, storing the compressed representation in index bits in a color buffer and marking the cache lines as evicted. The media may further store instructions to perform a sequence including where the palette is a set of N colors, and each sample uses N combinations to point into the palette, where the sum of the storage of the palette and the sum of all samples' combinations can be stored in index bits of a tile. The media may further store instructions to perform a sequence including selecting a tile, including cache lines, for eviction from a color buffer cache and checking index bits in the color cache to determine a number of planes in the color buffer. The media may further store instructions to perform a sequence including checking to determine whether all of said planes are in a color buffer cache. The media may further store instructions to perform a sequence including only attempting to compress if all of the planes are in the color buffer cache. The media may further store instructions to perform a sequence including using a different compression if all of the planes are not in the cache. The media may further store instructions to perform a sequence including using a different compression if the bit budget is not met. The media may further store instructions to perform a sequence including using three index bits to indicate which single color all samples point to, which pair of colors all samples point to and that the samples point to all three colors. The media may further store instructions to perform a sequence including using 64 bits to store pixels and if these bits are not sufficient then using a different compression method or no compression.

In another example embodiment may be an apparatus comprising a hardware processor to attempt to compress cache lines using a color palette, check whether the palette compression meets a bit budget, and if so, store an indicator in a control surface that palette compressor was used, storing the compressed representation in index bits in a color buffer and marking the cache lines as evicted, and a storage coupled to said processor. The apparatus may include said processor to where the palette is a set of N colors, and each sample uses N combinations to point into the palette, where the sum of the storage of the palette and the sum of all samples' combinations can be stored in index bits of a tile. The apparatus may include said processor to select a tile, including cache lines, for eviction from a color buffer cache and check index bits in the color cache to determine a number of planes in the color buffer. The apparatus may include said processor to check to determine whether all of said planes are in a color buffer cache. The apparatus may include said processor to only attempt to compress if all of the planes are in the color buffer cache. The apparatus may include said processor to use a different compression if all of the planes are not in the cache. The apparatus may include said processor to use a different compression if the bit budget is not met. The apparatus may include said processor to use three index bits to indicate which single color all samples point to, which pair of colors all samples point to and that the samples point to all three colors. The apparatus may include said processor to use 64 bits to store pixels and if these bits are not sufficient then using a different compression method or no compression. The apparatus may include said processor to use a four color palette.

The graphics processing techniques described herein may be implemented in various hardware architectures. For example, graphics functionality may be integrated within a chipset. Alternatively, a discrete graphics processor may be used. As still another embodiment, the graphics functions may be implemented by a general purpose processor, including a multicore processor.

References throughout this specification to "one embodiment" or "an embodiment" mean that a particular feature, structure, or characteristic described in connection with the embodiment is included in at least one implementation encompassed within the present disclosure. Thus, appearances of the phrase "one embodiment" or "in an embodiment" are not necessarily referring to the same embodiment. Furthermore, the particular features, structures, or characteristics may be instituted in other suitable forms other than the particular embodiment illustrated and all such forms may be encompassed within the claims of the present application.

While a limited number of embodiments have been described, those skilled in the art will appreciate numerous modifications and variations therefrom. It is intended that the appended claims cover all such modifications and variations as fall within the true spirit and scope of this disclosure.

What is claimed is:

1. A method comprising:

using palette compression in a hardware processor, compressing cache lines using a color palette;

checking whether the palette compression meets a bit budget; and

if so, storing an indicator in a control surface that palette compression was used, storing the color palette compressed representation in index bits in a color buffer and marking the cache lines as evicted.

2. The method of claim **1**, where the palette is a set of N colors, and using a plurality of samples with N combinations to point to a color in the palette, where N is a positive integer.

3. The method of claim **2**, where the palette has 2^n colors, and using n palette index bits to point into the palette, and where the sum of the storage of the palette and the number of palette index bits for all samples in the tile is less than or equal to the sum of the index bits of the tile, wherein n is a positive integer.

4. The method of claim **2**, wherein the palette is 2^n colors, and each sample uses less than n palette index bits to point into the palette, where n is a positive integer.

5. The method of claim **1** including selecting a tile, including cache lines, for eviction from a color buffer cache and checking index bits in the color cache to determine a number of planes in the color buffer and checking to determine whether all of said planes are in a color buffer cache.

6. The method of claim **5** including only attempting to compress if all of the planes are in the color buffer cache.

7. The method of claim **6** including using a different compression if all of the planes are not in the cache.

8. The method of claim **1** including using a different compression if the bit budget is not met.

9. The method of claim **1** including, for 4× multi-sampling anti-aliasing, storing a 2-color palette, and 1 palette index bit per sample.

10. The method of claim **1** including, for 4× multi-sampling anti-aliasing, storing 3 status bits per pixel indicating how the palette indices are encoded.

11. The method of claim **1** including, for 8× multi-sampling anti-aliasing, storing a 4-color palette, and 2 palette index bits per sample.

12. The method of claim **1** including, for 8× multi-sampling anti-aliasing, storing 4 status bits per pixel indicating how the palette indices are encoded.

13. One or more non-transitory computer readable media storing instructions to execute a sequence in a hardware processor comprising:

using palette compression to compress cache lines using a color palette;

checking whether the palette compression meets a bit budget; and

if so, storing an indicator in a control surface that palette compression was used, storing the color palette compressed representation in index bits in a color buffer and marking the cache lines as evicted.

14. The media of claim **13**, where the palette is a set of N colors, and using a plurality of samples with N combinations to point to a color in the palette, where N is a positive integer.

15. The media of claim **13**, said sequence including selecting a tile, including cache lines, for eviction from a color buffer cache and checking index bits in the color cache to determine a number of planes in the color buffer.

16. The media of claim **15**, said sequence including checking to determine whether all of said planes are in a color buffer cache.

17. The media of claim **16**, said sequence including only attempting to compress if all of the planes are in the color buffer cache.

18. The media of claim **17**, said sequence including using a different compression if all of the planes are not in the cache.

19. The media of claim **13**, said sequence including using a different compression if the bit budget is not met.

20. The media of claim **13**, said sequence including using three index bits to indicate which single color all samples point to, which pair of colors all samples point to and that the samples point to all three colors.

21. The media of claim **20**, said sequence including using 64 bits to store pixels and if these bits are not sufficient then using a different compression method or no compression.

22. An apparatus comprising:

a hardware processor to use palette compression to compress cache lines using a color palette, check whether the palette compression meets a bit budget, and if so, store an indicator in a control surface that palette compression was used, storing the color palette compressed representation in index bits in a color buffer and marking the cache lines as evicted; and

a storage coupled to said processor.

23. The apparatus of claim **22** where the palette is a set of N colors, and said processor to use a plurality of samples with N combinations to point to a color in the palette, where N is a positive integer.

24. The apparatus of claim **22**, said processor to select a tile, including cache lines, for eviction from a color buffer cache and check index bits in the color cache to determine a number of planes in the color buffer.

25. The apparatus of claim **24**, said processor to check to determine whether all of said planes are in a color buffer cache.

26. The apparatus of claim **25**, said processor to only attempt to compress if all of the planes are in the color buffer cache.

27. The apparatus of claim **26**, said processor to use a different compression if all of the planes are not in the cache.

28. The apparatus of claim **22**, said processor to use a different compression if the bit budget is not met.

29. The apparatus of claim **22**, said processor to use three index bits to indicate which single color all samples point to, which pair of colors all samples point to and that the samples point to all three colors.

30. The apparatus of claim **29**, said processor to use 64 bits to store pixels and if these bits are not sufficient then using a different compression method or no compression.

* * * * *